25 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Effective and Efficient Process Engine Evaluation

Simon Harrer

University of Bamberg Press

25 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Band 25

University of Bamberg Press
2017

Effective and Efficient Process Engine Evaluation

Simon Harrer

University of Bamberg Press
2017

Bibliographische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Informationen sind im Internet über http://dnb.d-nb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Guido Wirtz, Otto-Friedrich-Universität Bamberg
2. Gutachter: Prof. Dr. Dr. h. c. Frank Leymann, Universität Stuttgart
Tag der mündlichen Prüfung: 17.07.2017

Herstellung und Druck: docupoint, Magdeburg
Umschlaggestaltung: University of Bamberg Press, Larissa Günther

ISSN: 1867-7401
ISBN: 978-3-86309-503-1 (Druckausgabe)
eISBN: 978-3-86309-504-8 (Online-Ausgabe)
URN: urn:nbn:de:bvb:473-opus4-496333
DOI: http://dx.doi.org/10.20378/irbo-49633

To Martina

Acknowledgments

Finally.
It is done.
After six years.
I am happy and relieved.

However, this would not have been possible without numerous helpful, kind, intelligent, supportive, and critical friends, reviewers, colleagues, supervisors, family members, students, and all others not listed before. This is a big thank you to all of you.

The first step towards this dissertation was laid in 2010 when Prof. Dr. Guido Wirtz offered me a position as a research assistant in his Distributed Systems Group, which I gladly accepted. As my doctoral advisor, he always took the time to discuss my work despite his other obligations due to being dean or vice president, was quick and uncomplicated in approving my numerous application forms for the business trips (i.e., "Dienstreiseanträge") despite them being expensive and, well, numerous, and gave me the freedom to work on my research. For all of his support, I am deeply grateful. Apart from my advisor, I would like to thank the rest of my thesis committee: Prof. Dr. Sven Overhage and Prof. Dr. Udo Krieger, for their insightful comments and encouragement, but also for the hard questions which gave me the incentive to widen my research perspectives. Furthermore, I am deeply grateful that Prof. Dr. Dr. h. c. Frank Leymann has offered me to be the second assessor of this work. His offer actually marked a turning point of my research, as it motivated me to proceed further despite my doubts.

I had the privilege to work in an extremely supportive environment within the Distributed Systems Group. In my mind, I did not only have supportive colleagues but also very good friends. Cornelia Schecher, the good soul and backbone of the group, saved me countless hours by being the proxy to the administration and was always open for both small and deep talk during the coffee break. The joint creation of the tool betsy with Jörg Lenhard marked the start of this work in 2012. Two years later, Matthias Geiger joined our efforts by giving betsy the kick in the direction of BPMN. My dissertation is highly tied to the joint work with both of them. Stefan Kolb, although not collaborating with me on the betsy tool, has always helped out readily when needed. My friend Linus Dietz, who joined this group for the last year of my employment, joined the overall helpfulness of the group quickly. I am deeply in debt for their support during the whole time at the Distributed Systems Group and

beyond. My scientific career started with the diploma thesis I wrote supervised by Andreas Schönberger, for which I am grateful. He pushed me towards high quality research, and helped me to get my first conference publication. Moreover, I want to thank my other colleagues Mostafa Madiesh and Christian Wilms for their support in the early phases of this thesis.

During the six years, I have done a lot of teaching as well, including the supervision of projects, seminars, and theses. I had the pleasure to work with a lot of students on topics related to my dissertation, which resulted in either conference/workshop publications, technical reports, or open source tools – and in new friendships as well. I would like to sincerely thank the following persons for working hard on particular items of this thesis together with me: Lara Aubele, David Bimamisa, Christoph Bröker, Mathias Casar, Rolf Fakesch, Mathias Müller, Christian Preißinger, Cedric Röck, Stephan Schuberth, and Andreas Vorndran.

I also want to thank all those who criticized this work in some sort along the way to completion, especially the reviewers who gave their time reading and checking my submitted workshop, conference or journal papers. You gave me the opportunity to improve myself and this work.

I have gratefully benefited from all the discussions with my good friend Oliver Kopp, but also from his numerous constructive comments and persistent warm encouragements to keep going. You were incredibly helpful to secure the quality of this work! With Tammo van Lessen, I was able to publish a paper, which profited highly from his multiple roles as maintainer of an open source project I benchmark in this work, as a consultant working in industry, and his background in academic research. I am grateful to have worked with both Rix Groenboom and Faris Nizamic, and having found two great friends in the Netherlands (and was obliged to visit them there, of course). To Mazedur Rahman, I am in debt as he has shown my wife and me around the Sony Headquarters, and for the interesting discussions about the application of Docker. I also fondly remember the collaboration with Vincenzo Ferme and his hospitality in Lugano and, especially, Como. To Marigianna Skouradaki, I am also grateful for the collaboration and exchange on workflow benchmarking over the last few years.

I am in debt towards Babette Schaible who managed the paper work along the way. Thanks also to Thomas Benker, who started at the same time as myself, for the indirect motivation as he progressed in his dissertation often quicker than I did. A thank you also to the JabRef developers and the community – you always gave me an excuse to work on my programming skills for life after my dissertation. Furthermore, I want to thank all the other students who helped me in my teaching efforts: Christian Preißinger, Michael Träger, and Hendrik Cech. You freed up some time, so I could work on this dissertation. A big thank you also goes to the proof readers: Matthias Geiger, Martina Harrer, Oliver Kopp, Philipp Neugebauer, and Michael Oberparleiter.

Last but not least, I owe my deepest gratitude to my wife Martina, as without her support, love, and sometimes a little bit of pressure, I would have quit at some point in time, I am sure. Another anchor of support has been my whole family, including my grandparents Oma and Opa, my parents Ingrid and Erhard, my in-laws Maria and Peter, and my two sisters Anne and Franziska – they were and are always there for me (even Opa from far above). Thank you, Anne for providing me a writing hide out in Passau, where a lot of these pages were crafted. Especially, when I had motivational issues, the discussions with my father have kept me on track – I do remember them fondly. And I am glad I did have a phone flatrate.

Kurzfassung

Geschäftsprozesse sind in der Industrie allgegenwärtig und Geschäftsprozessmanagement daher ein wichtiger Baustein in Unternehmensabläufen. Prozessausführungsumgebungen erlauben die automatische Ausführung von Geschäftsprozessen. Die zwei bekanntesten standardisierten Sprachen, um Geschäftsprozesse zu modellieren, sind die Web Services Business Process Execution Language 2.0 (BPEL) und Business Process Model and Notation 2.0 (BPMN). Zur Auswahl stehen für beide eine Vielzahl von Ausführungsumgebungen und somit besteht die Qual der Wahl: Welche Ausführungsumgebung erfüllt die Anforderungen am besten? Eine rationale Auswahl wird durch das Fehlen von objektiven, reproduzierbaren und gesicherten Informationen über die Qualität solcher Ausführungsumgebungen verhindert. Dies kann zu unfundierten und unausgereiften Entscheidungen und diese wiederum zu hohen Kosten führen. Problem

Diese Arbeit stellt eine effiziente und effektive Benchmarkinglösung vor, um die notwendigen Informationen für rationale Entscheidungen aufdecken zu können. Das Fundament besteht aus einer Abstraktionsschicht und einer Benchmarkingsprache für Prozessausführungsumgebungen. Die Abstraktionsschicht stellt eine uniforme API bereit um mit jedem möglichen System in gleicher Weise zu interagieren und die Benchmarkingsprache ermöglicht es Benchmarks in einer kompakten, abgeschlossenen und interpretierbaren domänenspezifischen Sprache darzustellen. Das Benchmarkingrahmenwerk für Prozessausführungsumgebungen führt Benchmarks, die in dieser Sprache repräsentiert sind, auf Ausführungsumgebungen aus, welche die Abstraktionsschicht implementieren. Die erzeugten Benchmarkingergebnisse werden durch ein interaktives Dashboard visualisiert und somit Entscheidern zugänglich gemacht. Aufbauend auf dem Benchmarkingrahmenwerk verwendet das effiziente Benchmarkingrahmenwerk von Prozessausführungsumgebungen virtuelle Maschinen, um Testisolierung zu erreichen und um die Zeit bis zum Vorliegen der Ergebnisse zu reduzieren. Der dabei entstandene zusätzliche Verwaltungsaufwand, der sich durch die Wiederherstellung von Snapshots ergibt, bleibt akzeptabel. Aufbauend auf den gewonnen Erfahrungen werden acht Herausforderungen im Bereich des Benchmarkings von Prozessausführungsumgebungen identifiziert, die wiederum in 21 Patternkandidaten resultieren. Lösung

Die Ergebnisse zeigen, dass der beschriebene Ansatz sowohl effektiv als auch effizient ist. Effektiv, da eine Reihe von Qualitätscharakteristiken des ISO/IEC 25010 Produktqualitätsmodells von sowohl BPEL-basierten als auch BPMN-basierten Prozessausführungsumgebungen bestimmt werden können. Ef- Ergebnis

fizient, da das Benchmarking von Prozessausführungsumgebungen vollständig automatisiert und die Vorteile der Virtualisierung für eine noch höhere Ausführungseffizienz und Testisolation genutzt wurden. Dadurch wird die Hürde, gute Benchmarks zu erstellen, signifikant herabgesetzt. Dies ermöglicht die Bewertung von Prozessausführungsumgebungen und erleichtert somit die dazu in Bezug stehenden rationalen Auswahlentscheidungen.

Abstract

Business processes have become ubiquitous in industry today. They form the main ingredient of business process management. The two most prominent standardized languages to model business processes are Web Services Business Process Execution Language 2.0 (BPEL) and Business Process Model and Notation 2.0 (BPMN). Business process engines allow for automatic execution of business processes. There is a plethora of business process engines available, and thus, one has the agony of choice: which process engine fits the demands the best? The lack of objective, reproducible, and ascertained information about the quality of such process engines makes rational choices very difficult. This can lead to baseless and premature decisions that may result in higher long term costs. Problem

This work provides an effective and efficient benchmarking solution to reveal the necessary information to allow making rational decisions. The foundation comprises an abstraction layer for process engines that provides a uniform API to interact with any engine similarly and a benchmark language for process engines to represent benchmarks in a concise, self-contained, and interpretable domain-specific language. A benchmark framework for process engines performs benchmarks represented in this language on engines implementing the abstraction layer. The produced benchmark results are visualized and made available for decision makers via a public interactive dashboard. On top of that, the efficient benchmark framework uses virtual machines to improve test isolation and reduce "time to result" by snapshot restoration accepting a management overhead. Based on the gained experience, eight challenges faced in process engine benchmarking are identified, resulting in 21 process engine benchmarking. Solution

Results show that this approach is both effective and efficient. Effective because it covers four BPEL-based and another four BPMN-based benchmarks which cover half of the quality characteristics defined by the ISO/IEC 25010 product quality model. Efficient because it fully automates the benchmarking of process engines and can leverage virtualization for an even higher execution efficiency. With this approach, the barrier for creating good benchmarks is significantly lowered. This allows decision makers to consistently evaluate process engines and, thus, makes rational decisions for the corresponding selection possible. Result

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

List of Abbreviations

AHP Analytic Hierarchy Process
ANP Analytic Network Process
API Application Programming Interface
APPDATA Application-Specific Data
betsy BPEL/BPMN Engine Test System
BI Business Intelligence
BLCR Berkely Lab Checkpoint/Restart
BPaaS Business Process as a Service
BPEL Web Services Business Process Execution Language 2.0
BPMN MIWG BPMN Model Interchange Working Group
BPMN Business Process Model and Notation 2.0
BPMS Business Process Management System
BPM Business Process Management
BPRD Business Process Archive Descriptor
BPR Business Process Archive
C2FM Capability to Feature Method
CAMP Cloud Application Management for Platforms
CDATA Character Data
cgroup control group
CI Continuous Integration
CLI Command Line Interface
COTS Commercial Off-The-Shelf
CPU Central Processing Unit
CSS Cascading Stylesheets
DMTF Distributed Management Task Force
DNS Domain Name System
DOM Document Object Model
DSG Distributed Systems Group
DSL Domain-Specific Language
DSS Decision Support System
DWH Data Warehouse
ELECTRE Elimination and Choice Expressing Reality
ePEBWORK Efficient Process Engine Benchmark Framework
ERP Enterprise Resource Planning
ES6 ECMAScript 6
ESB Enterprise Service Bus

ETL Extract-Transform-Load
F2TM Feature to Test Method
FCT fault, compensation, and termination
FR Functional Requirement
FS File System
GB Giga Byte
GUI Graphical User Interface
HDD hard disk drive
HTML5 Hypertext Markup Language 5
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
IaaS Infrastructure as a Service
ID Identifier
IEC International Electrotechnical Commission
IEEE Institute of Electrical and Electronics Engineers
IoT Internet of Things
IO Input/Output
ISO International Standards Organization
JAR Java Archive
JAX-WS Java API for XML-Based Web Services
JAXB Java Architecture for XML Binding
JBI Java Business Integration
JEE Java Platform, Enterprise Edition, also Java EE, previously J2EE
JMS Java Messaging System
JRE Java Runtime Environment
JRE Java Runtime Environment
JSON JavaScript Object Notation
JS JavaScript
JVM Java Virtual Machine
JPDL jBPM Process Definition Language
KO Knock-Out
KPI Key Performance Indicator
KVM Kernel-based VM
LTS long-term support
MADM Multi Attribute Decision Making
MCDM Multi Criteria Decision Making
MEP Message Exchange Pattern
MIME Multipurpose Internet Mail Extensions
MODM Multi Object Decision Making
MVVM Model-View-ViewModel
NAT Network Address Translation
NFR Nonfunctional Requirement
OASIS Organization for the Advancement of Structured Information Standards

OCCI Open Cloud Computing Interface
OCI Open Container Initiative
OED Oxford English Dictionary
OMG Object Management Group
OS Operating System
OVA Open Virtual Appliance
OVF Open Virtualization Format
PaaS Platform as a Service
PEAL-WS PEAL Web Services
PEAL Process Engine Abstraction Layer
PEBDASH Process Engine Benchmarking Interactive Dashboard
PEBL Process Engine Benchmark Language
PEBPATT Process Engine Benchmarking Pattern Candidates
PEBWORK Process Engine Benchmark Framework
PNG Portable Network Graphics
PoC Proof of Concept
PVM Process Virtual Machine
QName Qualified Name
QoS Quality of Service
RAM random-access memory
REST Representational State Transfer
RFC Request for Comments
RfI Request for Information
RfP Request for Proposal
RUSP Ready to Use Software Product
SA Static Analysis
SaaS Software as a Service
Sass Syntactically Awesome Style Sheets
SOAP SOAP
SOA Service-Oriented Architecture
SPDX Software Package Data Exchange
SPEC Standard Performance Evaluation Corporation
SQL Structured Query Language
SQuaRE Systems and software Quality Requirements and Evaluation
SSD solid-state drive
SSH Secure Shell
SVM System Virtual Machine
TCK Test Compatibility Kit
TCP Transmission Control Protocol
TEC Technical Evaluation Centers
TFB TechEmpower Framework Benchmarks
TOPSIS Technique for Order Performance by Similarity to Ideal Solution
TOSCA Topology and Orchestration Specification for Cloud Applications

TPC-VMS TPC Virtual Measurement Single System Specification
TPC Transaction Processing Performance Council
UBML Uniform BPEL Management Layer
UI User Interface
UML Unified Modeling Language
URI Uniform Resource Identifier
URL Uniform Resource Locator
vbetsy virtualization-enabled betsy
VMM Virtual Machine Monitor
VMTH Virtual Machine Test Harness
VM Virtual Machine
WAPI Workflow APIs and Interchange Formats
WAR Web Archive
WfMC Workflow Management Coalition
WfMS Workflow Management System
WfM Workflow Management
WIP Work in Progress
WPM Weighted Product Model
WS-I Web Services-Interoperability Organization, part of OASIS since 2010
WSDL Web Service Description Language
WSIF Web Services Invocation Framework
WSM Weighted Sum Model
WSO2 BPS WSO2 Business Process Server
WS Web Service
XML eXtensible Markup Language
XPath XML Path Language
XPDL XML Process Definition Language
XSD XML Schema Definition
XSLT Extensible Stylesheet Language Transformations
XSL Extensible Stylesheet Language
YCSB Yahoo! Cloud Serving Benchmark
ZIP ZIP Archive

Part I.

Background and Problem Identification

Every moment you are open, as a humble student, you are surrounded with infinite possibilities of choice.

Bryant McGill

1. Introduction

This chapter introduces and motivates this work. First, the context is described in Section 1.1 followed by the problem statement in Section 1.2. The approach that presents the idea for a solution to the previously posed problem is described in Section 1.3. The used research method is described in Section 1.4, and the contributions that are produced by applying this research method are detailed in Section 1.5. Last, the remaining structure of the whole thesis is given in Section 1.6.

1.1. Context

Business Processes

The age of *processes* has arrived and is at its peak [53]. Although data (e.g., orders, incidents, or invoices) are still important, the focus lies on the processes (e.g., order handling, incident reporting, or invoice sending processes) instead. Business processes comprise activities that are conducted to achieve a business goal [156]. They can stretch either within an organization (i.e., orchestrations) or between organizations (i.e., choreographies) [44, 197]. Furthermore, business processes can have varying degrees of automation, repetition, and structuring [278]. Weske [278] defines a business process as follows:

> **Definition 1.1 (Business Process)**
> *"A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations."* [278, p. 5]

Business Process Management

Today, (business) processes are everywhere [73, 244]. Moreover, they have to be managed and constantly improved. Managing such processes does not only cover how to represent, but also how to design, enact, analyze, and improve them [278]. This discipline is called Business Process Management (BPM), and the constant optimization of processes that is standard practice in the industry is known as the BPM lifecycle [261]. According to Gartner Inc. [78], BPM is central for Business-IT alignment, and for an effective and efficient organization [278]. BPM is defined by Weske [278] as follows:

> **Definition 1.2 (Business Process Management)**
> *"Business process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes."* [278, p. 5]

Business Process Management System

Today, such business processes are not enacted manually, but automatically as this ensures a plethora of organizational benefits [278]. This can be done through implementing a software system that mirrors the behavior of one or more business processes. One possibility is to implement the business processes using a programming language such as Java. This would be similar to use an object-oriented programming language in the field of data persistence. There, an impedance mismatch was discovered [38]. A better solution is to automate the processes through a generic software system that can execute instances of previously defined business processes *natively* without the need for any conceptual mapping. Such systems are called Business Process Management Systems (BPMSs) [53, 125, 278] and are defined as follows by Weske [278]:

> **Definition 1.3 (Business Process Management System)**
> *"A business process management system is a generic software system that is driven by explicit process representations to coordinate the enactment of business processes."* [278, p. 5]

Process Engines

The terms BPMS and Workflow Management System (WfMS) are used synonymously, similar to Reijers [206]. A *process engine* is solely responsible for enacting processes leaving aside all other functionality [278], similar to the *workflow engine* of WfMSs [154, 279]. This work concentrates solely on process enactment. Hence, the term process engine, which is considered the heart of a BPMS, is used instead of BPMS in the following to account for that specific focus. Concentrating on process engines also leaves the business process modeling tools out of scope. The definition by Weske [278] is used:

> **Definition 1.4 (Process Engine)**
> *"The process engine is responsible for instantiating and controlling the execution of business processes. It is the core component of a business process management system."* [278, p. 121]

RUSP

Such generic software systems are known as Ready to Use Software Products (RUSPs) (or under the older term Commercial Off-The-Shelf (COTS) Software Products) as they are not developed per project, but developed once by a vendor and then ready to be used in any suitable project [117]. The definition according to the ISO/IEC standard is given in the following. Similar to other generic software such as the text processor Microsoft Word, they are installed, configured, used, and extended through plugins. Such RUSPs are only customized through configuration and extension, not through modifying

the underlying source code [117]. The explicit but unsubstantiated exclusion of open source products from being RUSPs in the ISO/IEC 25051 standard [117] is questionable. Especially regarding the large and complex process engines, it does not really matter whether one has access to the source code or not, as it is not feasible for the vast majority of the users to modify the software. Hence, the definition for RUSP in the following is also applied for complex and large open source software.

> **Definition 1.5 (Ready to Use Software Product)**
> *"[A Ready to Use Software Product] is a software product available for any user, at cost or not, and use without the need to conduct development activities."* [117, p. 3]

Standards and Languages

Business processes are represented in process models using process languages. Since the rise of processes, at least 19 different business process modeling languages have emerged [171]. The two most notable ones are defined as part of the OASIS standard Web Services Business Process Execution Language 2.0 (BPEL) [181] and the ISO standard[1] Business Process Model and Notation 2.0 (BPMN) 2.0 [189]. Using these languages, business processes can be modeled in a way so that they are directly executable on supporting process engines by means of business process instances of business process models.

> **Definition 1.6 (Business Process Instance)**
> *"A business process instance represents a concrete case in the operational business of a company, [...] [e]ach business process model acts as a blueprint for a set of business process instances [...]."* [278, p. 7]

Business Process Example

An example of a business process is given in Figure 1.1. It has been taken from the BPMN ISO/IEC [115, p. 168] standard and shows a procurement process from the perspective of a buyer. The buyer has to perform several tasks such as *handle quotations*, *handle order*, and *handle shipment* but needs his superior for the task *approve order* and a colleague for the task *review order* which implements the four-eyes principle [115].

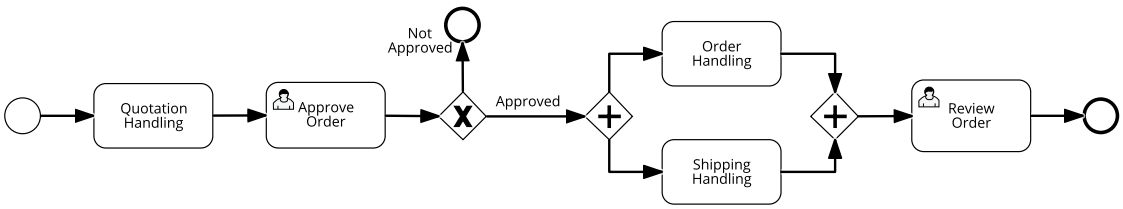


Figure 1.1.: Procurement process example using BPMN [115, p. 168]

[1] The BPMN standard has originally been created under the Object Management Group (OMG), but later has been converted to an International Standards Organization (ISO) standard.

Plethora of Standard-based Process Engines

Within the field of BPM, there are numerous vendors [191] available. Looking at the standard-based process languages, in particular, 74 implementers of the BPMN specification [190] can be found, and a list of BPMN process engines [283] is available as well. Moreover, there are more than ten BPEL engines available on the market according to Wikipedia [282]. Other listings of process engines are available as well (e.g., an overview of open source workflow engines [119]). Among BPEL and BPMN process engines, there are proprietary as well as popular open source engines available for both standard-based process languages. In summary, there are a plethora of implementations available per standard-based process language which compete with each other in enacting instances of process models defined using the respective language. The term process engine is used according to the following definition of a standard-based process engine throughout the remainder of this work.

Definition 1.7 (Standard-based Process Engine)
A process engine that enacts processes represented in process models using a standard-based process language.

Portability and Scope

In theory, process models defined according to the standard-based process language can be migrated from one standard-supporting process engine to another one. Since vendors want to distinguish their products from the competition, it is expected that these engines do vary greatly [56] and that can be observed as well [90, 237]. Concentrating on standard-based process engines excludes process engines that only support a single process language which is not standardized, e.g., Windows Workflow [30], as if there is only one alternative, there is no agony of choice. Put differently, this work concentrates on standard-based process languages for which there are multiple implementations (i.e., standard-based process engines) available. This does not limit the contributions of this work as BPEL [181] and BPMN [115], the two most popular process languages, are standard-based [129, 171], and for them, more than ten process engines per language have emerged.

Adoption and Market

In the latest report from Forrester Research[2], Richardson et al. [208] estimate that the BPM market will have reached $6 billion in 2015. The latest Gartner report *Magic Quadrant for Intelligent Business Process Management Suites* [54] estimates a market for BPMSs at $2.7 billion in 2015. According to the BPTrends[3] study about the *State of Business Process Management* by Harmon and Wolf [92], 19% of the surveyed companies say that BPMSs are the most important tools in 2015. Furthermore, 19% plan to buy such a BPMS in 2016, resulting in the need to determine *which* of the available BPMS to select. Hence, the selection and comparison of the available process engines is an actual problem in industry because the decision is imminent shortly.

[2] `http://forrester.com/research/`, visited 2017-3-31
[3] `http://www.bptrends.com/`, visited 2017-3-31

Summary To sum up, different engines (i.e., alternatives) enact processes modeled using standard-based process languages. Shortly, companies need to overcome the *agony of choice* in their move towards a higher adoption of BPM in a large market. Hence, this leads to and supports hypothesis H1.

> **Hypothesis H1** *Today, business procedures are modeled in automatable business processes. Process engines can help as they allow executing previously modeled business processes directly. For the popular business process standards, there are too many different process engines available, resulting in an agony of choice.*

1.2. Problem Statement

Multi Criteria Decision Making The situation of choosing one out of a set of competing and discrete alternatives according to multiple different criteria is known as Multi Criteria Decision Making (MCDM) [254]. Applied to our context, process engines compete which each other to enact process models based on a standard-based process language. To make an informed, rational, and accountable decision, the process engines (i.e., the alternatives) are compared with each other based on how well they fulfill project-dependent requirements (i.e., the criteria).

> **Definition 1.8 (Multi Criteria Decision Making)**
> *"[Multi criteria decision making answers the question:] given a set of alternatives and a set of decision criteria, then what is the best alternative?"* [254, p. XXV]

Decision Making Steps A variety of different formal methods is available to make a rational decision. Although there are general decision making methods such as the Analytic Hierarchy Process (AHP) by Saaty [222], there are also specific ones targeted at the evaluation and selection of RUSPs by ISO/IEC [114], Lawlis et al. [145], Tarawneh et al. [247]. Even for evaluating BPMSs, there is an approach put forward by Delgado et al. [45]. They share the same general decision making process comprising the problem definition, requirements identification, goal establishment, alternative identification, evaluation criteria development, decision making method selection, alternative evaluation by criteria, and final solution validation [8]. Although the decision making methods vary, they all need to evaluate each alternative against previously specified criteria using quantitative or qualitative methods. In the example evaluation process for COTS products [114], the step is called "[e]valuate software products based on external evaluation results, product documentation, product operating experience, product prototyping, [and] other product evaluation methods." [114, Figure 3 on p. 31] The outcome of those evaluations influences the quality of the outcome of the selection. The more trusted information (i.e., objective,

reproducible, and ascertained information) is available, the more profound the decision will be. Put differently, if the information cannot be revealed or is not available, resulting in uncertainty, effective decision making is aggravated [157]. In this work, the focus is on providing a method to reveal the quality of process engines so that uncertainty in decision making of selecting process engines is reduced.

Product Quality Model

The quality of process engines is understood according to the *product quality model* defined in the ISO/IEC 25010:2011 standard [113]. The product quality model forms a hierarchic structure with quality characteristics and sub-characteristics to categorize quality attributes (i.e., measurable quality properties). Within this standard, the product quality model comprises eight quality characteristics ranging from *functional suitability* and *performance efficiency* to *usability* and *resilience*. Using this quality model, the quality requirements as well as the actual quality properties of process engines can be categorized and expressed. The quality model has the advantage that it covers both, functional and nonfunctional aspects, and therefore provides a holistic view of the technical quality of process engines. It solely focuses on the technical aspects of the product, leaving aside nontechnical characteristics such as pricing, licensing, and support contract conditions [45].

Definition 1.9 (Software Quality Characteristic)
"[A software quality characteristic is a] category of software quality attributes that bears on software quality." [113, p. 19]

Definition 1.10 (Quality Attribute)
"[A quality attribute is an] inherent property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means." [113, p. 18]

Quality Characteristics and KO Criteria

The quality characteristic *functional suitability* is the core quality characteristic as it corresponds to functional aspects [113]. This quality characteristic along with its sub-characteristics mostly represent Knock-Out (KO) criteria (i.e., hygiene factors by Herzberg [107]). If they are not supported, the process engine cannot be considered at all, and the evaluation would indicate a "red flag." They must be supported because they are necessary to execute process instances of the process models that are or will be created within that project. Mature software products should already fulfill Knock-Out (KO) criteria. For instance, if the required functional attribute for a BPMN process engine is that the exclusive gateway is necessary and a particular engine does not support it, the criterion is a KO criterion as without it the project cannot be implemented using that process engine. The other seven quality characteristics match with nonfunctional aspects, e.g., *performance efficiency* or *resilience* [113]. They express the Quality of Service (QoS) when making use of the *functional suitability* characteristics. Nowadays, some of them are seen as KO characteristics

as well, e.g., robustness and resilience upon failure [153]. For instance, if a single process instance running into an infinite loop can crash the whole process engine, this probably renders the process engine unusable for many use cases which require a stable and working process engine. Not every nonfunctional attribute, however, is seen as a KO criterion. If, for instance, a BPEL process engine performs the parallel for each loop sequentially instead of in parallel as requested, it may still be ok as long as the engine performs well enough despite that behavior. KO criteria are not useful for MCDM methods that work by weighting and prioritizing features, comparing mature alternatives with each other. They can be seen as the first evaluation phase before a more sophisticated selection method such as AHP is applied.

Information

In summary, applying decision making methods requires the availability of information about the quality of the possible alternatives (i.e., process engines). Using the terminology of the ISO/International Electrotechnical Commission (IEC) 25010 standard [113], it requires *external measure of software quality*. If such information is not available, or not of the necessary quality, the quality of the decision suffers. Hence, this leads to and supports hypothesis H2.

Hypothesis H2 *Objective, reproducible, and ascertained information about the quality of process engines is the foundation for a methodical comparison and selection of the best fitting process engine for a specific set of quality requirements.*

Hidden Truth

The ultimate truth of the quality attributes of a process engine resides within the product itself. However, software is complex, hard to build, and hard to understand. One reason for this is that "[s]oftware is not very visible" [12, p. 1]. Its characteristics are hidden. Hence, revealing these characteristics helps in reducing the complexity of software, making them easier to build and understand. Revealing the quality characteristics of process engines would make them also easier to understand, and therefore compare and select.

Available Information

Today, information about the quality of process engines is already available. That information has been extracted from the products by different stakeholders [117]: the *vendor* or other *third parties* such as independent research institutes (e.g., Gartner[4] [54, 122, 230, 231], TEC[5] [248], Fraunhofer Institute[6] [1, 2], and Forrester[7]), IT consultancies (e.g., Capgemini Consulting[8] [226]), or researchers (e.g., Workflow Patterns Initiative [291], [289], and others [16, 41, 45, 237, 289]). The information about the quality of the process engines is presented in different artifacts: *product information*[9],

[4] `https://www.gartner.com/`, visited 2017-3-31
[5] `http://www.technologyevaluation.com/`, visited 2017-3-31
[6] `https://www.fraunhofer.de/en.html`, visited 2017-3-31
[7] `https://www.forrester.com`, visited 2017-3-31
[8] `https://www.capgemini-consulting.com/`, visited 2017-3-31
[9] E.g., `http://ode.apache.org/` or `https://camunda.org/`, visited 2017-3-31

product documentations[10], *studies* [1, 2, 54, 122, 159, 160, 226, 230, 231], *books* [152, 204], and *scientific papers* [16, 41, 45, 237, 289]. Each of them is evaluated regarding the quality of their provided information subsequently in the following.

Product Information and Documentations

Vendors have detailed knowledge about the characteristics of their products, and market them on their websites. Some vendors do provide a comparison to other engines, e.g., on the homepage of bpel-g[11], but the feature matrices typically show that the product of the vendor is the best, and therefore contain only limited value. A sign for that is that neither the homepage nor the documentation state what the engine cannot do or where its weaknesses are. Moreover, product information may not be telling the whole truth[12] or may be filtered through the marketing team in a way to make the presented information hard to use for comparisons. The documentation of a product does not suffer from being marketing material, but instead from its size, effectively hiding the relevant information in large unstructured text documents. It typically lacks a structured and comparable form of the product quality attributes. Moreover, the documentation may be out of sync with the product as both are two separate artifacts. A manual verification of the documentation, however, is not feasible.

Studies

Research institutes aim to draw pictures of the state-of-the-art to reveal trends. This ranges from comparing process engines on a high level [54, 122, 230, 231] and on a more detailed one [1, 2]. The high level studies are conducted using a standardized questionnaire that is answered by the vendors, and the more detailed studies via a one-day workshop with the vendors. Consequently, even the more detailed studies lack detail and relevance because a single day used in those studies is not enough time to reveal information of high detail and quality. Consultants that are tasked with a BPM project as Scheithauer and Klinnert [226] help clients for their selection of the appropriate process engine by conducting an evaluation of them, but their results are not disclosed, and are, therefore, not considered available.

Books

Domain experts and vendors write books about the available products, such as Lessen et al. [152], Rademakers [204]. For instance, Rademakers [204, pp. 398] provide a summary of the supported language features of the BPMN engine Activiti. Furthermore, a comparison of Activiti with other BPMN engines is available by Rademakers [204, pp. 7] as well. In contrast, Lessen et al. [152, Chapter 9] provide a description of three proprietary and one open source BPEL engine, but there is no comparison of their features. Both information is completely outdated by now, as the release of these books was over four years ago.

[10] E.g., `http://ode.apache.org/userguide/` or `https://docs.camunda.org/manual/7.6/`, visited 2017-3-31

[11] `https://code.google.com/p/bpel-g/wiki/BPELComparison`, visited 2017-3-31

[12] See the press release of the NVIDIA Corporation for putting out wrong information about their product at `http://blogs.nvidia.com/blog/2015/02/24/gtx-970/`, visited 2017-3-31.

Scientific Papers

Published and peer-reviewed scientific papers [16, 41, 79, 237, 289] contain results about the quality attributes of process engines. This includes the performance comparison of three BPMN process engines by Skouradaki et al. [237] and the performance evaluation of three BPEL process engines by Bianculli et al. [16]. Workflow patterns are suitable for comparing the quality of process engines (Section 2.2.3). Wohed et al. [289] present a pattern-based analysis of long outdated open source BPMSs. A more complete pattern-based evaluation [291] is available as well by the Workflow Patterns initiative. Papers that present the evaluation of modeling tools are related but not relevant as well [41, 79]. Due to long publication procedures, especially for journals, the results presented in scientific papers are outdated upon publication. Moreover, in the mentioned papers, only aggregated results are presented because of the page-restricting nature of peer-reviewed papers and more detailed results are missing as well.

Available Information Gap

The available information suffers from being irrelevant, unaccountable, unstructured, incomparable, and outdated. Although the vendor has the deepest knowledge about the product, he is biased because he wants to show his product in the best light possible, whereas the other more objective third parties lack the knowledge to publish relevant information. That situation aggravates any comparison of the quality between process engines from different vendors or between different versions of a process engine from a single vendor over time.

Coping with Uncertainty

Due to the lack of available information, there is uncertainty in making the right decision for a particular process engine. One of the three three reasons for uncertainty in decision making is incomplete information [157]. Although there are ways to cope with incomplete information, the first one is to conduct a thorough information search, applying the strategy of reducing uncertainty to complete the previously lacking information [157]. In reality, gathering that information is hard because the information is often unavailable, ambiguous, misleading, or worthless [58, 88, 157, 290]. This holds for the problem of this work as well. For instance, the label “standard compliant” raises expectations which often happen to remain unfulfilled [56].

Revealing Information

There are, however, ways to reveal information that is currently unavailable: through the initiative of the decision makers themselves [157]. Regarding Ready to Use Software Products, this is done through *Request for Information (RfI)* inquiries and *Proof of Concept (PoC)* demonstrations [117, 145].

Request for Information

The Request for Information (RfI)[13] questionnaires are collections of requirements of the decision maker encoded as questions which are answered by the vendor. Companies such as Technical Evaluation Centers (TEC) provide standardized questionnaires [249] and the corresponding vendor answers [248] for a plethora of process engines. Such standardized questionnaires are not tailored to the specific needs of the decision maker, which may be irrelevant. Moreover,

[13] Request for Proposal (RfP) is used synonymously in literature. In this work, however, only RfI is used for consistency.

the answers of the vendors are quickly outdated if the vendor does not put effort into keeping them up to date, similarly to the product documentation. What is more, the answers are given by the vendor. They are unverified and given in a way so that the product is presented in the best light. Also, it is not feasible to add every detailed question to the questionnaire, resulting in less and more abstract questions instead.

Proof of Concept

In PoC demonstrations (i.e., evaluation sessions [145]), the decision maker lets the vendors demonstrate specific use cases with their products. The decision maker then evaluates how well the product supports those use cases and extrapolates from that observation information about the quality of the process engine. Such demonstrations, however, require time from the decision maker and the vendor. Hence, because of that effort, the use cases are typically reduced to a minimal set which in turn limit the range of quality attributes they can cover. A PoC can also be used to verify some of the answers given by the vendor as part of a RfI questionnaire [159, 160].

Revealing Information Gap

As described, revealing information is hard. Applying the two described methods requires effort and the revealed information lacks depth. Moreover, available evaluation approaches [45, 145, 247] completely leave out how quality attributes can be revealed. Hence, this leads to and supports hypothesis H3.

Hypothesis H3 *Available information is not sufficient to methodically compare and select the best fitting process engine. In practice, this leads to unfunded and premature selection decisions that cause high long-term costs.*

1.3. Approach

Title

The title of this work is "Effective and Efficient Process Engine Evaluation." It comprises the constraints how the problem stated in Section 1.2 has to be solved. The solution that can evaluate process engines has to be *effective* and *efficient*. These constraints are defined as follows.

Definition 1.11 (Effectiveness)
"The degree to which something actually happens in the way it ought to happen." [87, p. 12]

Definition 1.12 (Efficiency)
"The degree to which a system or component performs its designated functions with minimum consumption of resources." [203, p. 30]

Effectiveness

Effectiveness in the context of process engine evaluation can be seen from a technical and a business perspective. The business perspective comprises the evaluation of process engines within industry. It includes capturing of

requirements in projects and checking whether an approach can reveal the relevant information for those specified requirements. This is out of scope of this work. Instead, the focus is put on the technical effectiveness which comprises the ability to extract relevant, objective, and ascertained information on the quality of process engines in a reproducible way and make them available such that they can be used as part of a selection decision. The extraction is a generic procedure for various different quality attributes. Hence, the definition of technical effectiveness is derived from the definition of effectiveness as follows:

> **Definition 1.13 (Technical Effectiveness)**
> *The degree to which something technically actually happens in the way it ought to happen.*

Efficiency

Efficiency is about minimizing the resource consumption. A resource is an abstract term for something that is not available indefinitely. To make it more precise, time is most crucial for process engine evaluation. In case an evaluation takes too long, it can block or delay dependent steps, e.g., decisions. In case, however, it can be performed quickly, it may even open up possibilities such as quick decisions and integration into Continuous Integration (CI) pipelines. Hence, instead of efficiency, the more precise term *execution efficiency* is used.

> **Definition 1.14 (Execution Efficiency)**
> *"The degree to which a system or component performs its designated functions with minimum consumption of time."* [203, p. 31]

Bench-marking

To support an effective and efficient process engine evaluation, it is proposed to apply automated benchmarking through tests. This follows the approach by the ISO/IEC standards family for selecting RUSP software [114, 117], but tailors it to process engines. Especially through automated benchmarking, it should be possible to reveal quality attributes in a traceable, ascertained, and reproducible way whereas the effort can be managed by appropriate languages, models, frameworks, and prototypes. It can even be combined with available information as it allows the decision maker to verify them. Hence, the following main research hypothesis is posed:

> **Hypothesis H4** *With automated benchmarking, it is possible to retrieve objective and comparable information on the quality characteristics of widely different process engines reproducibly and efficiently.*

Challenges

This raises a variety of challenges that makes it hard to test hypothesis H4 directly. Instead, hypothesis H4 is subdivided into six sub-hypotheses ranging from H4.1 to H4.6. Each of them focuses on a specific aspect and challenge of hypothesis H4, and is tested in separate chapters within Part II. The evaluation of hypothesis H4 itself is based on the testing of each sub-hypothesis. In

the following, the sub-hypotheses are detailed subsequently, starting with hypothesis H4.1.

Many-to-many Decision Makers and Different Products

Process engines vary widely in their quality characteristics, including the way they are used, e.g., installed, started, and process models deployed [150]. This poses the challenge of how to handle such widely differing software products in a variety of benchmarks. In reality, this is a many-to-many problem in which many decision makers have to compare many process engines. As a solution, a Process Engine Abstraction Layer (PEAL) is suggested that reduces the complexity so that the decision makers can use the abstraction layer and the vendors can implement the abstraction layer for their products. In other words, the following hypothesis is proposed:

Hypothesis H4.1 *A uniform interface is a suitable solution to interact with widely different process engines in a similar way.*

Clear and Concise Benchmark Representation

Another aspect of benchmarking is the challenge how to represent and express the benchmarks themselves. This is important as the benchmarks need to contain all necessary information on how to conduct them and to make their results reproducible. As a solution, a Domain-Specific Language (DSL) is proposed with an interchangeable serialization format. In this work, that language is called Process Engine Benchmark Language (PEBL). Such a Domain-Specific Language would act as a way to express the benchmarks and their results in a clear and concise way. In other words, the following hypothesis is proposed:

Hypothesis H4.2 *A domain-specific testing language is a suitable form to make quality criteria measurable.*

Automated Framework

Conducting a benchmark automatically is important to achieve objective and reproducible results, and to make benchmarking practical. This includes ensuring standard best-practices from the test domain such as test isolation. In this work, the Process Engine Benchmark Framework (PEBWORK) is proposed as a solution that can conduct benchmarks defined in PEBL on process engines supported by PEAL. Hence, the following hypothesis is proposed:

Hypothesis H4.3 *A benchmarking framework is a suitable means to reveal objective and well-founded information about process engines.*

Results Presentation

The benchmarks and their results contain the relevant information that a decision maker can use in the chosen decision making method such as AHP. The information, however, is hidden in the large set of data and needs to be presented in a way so that the information relevant for the decision maker can be found quickly. Moreover, the results need to be stored in a database so that other decision makers can make use of them as well without

actually conducting the benchmark by themselves. The evaluation can be performed at different points in time independent of the selection. As a solution, an interactive dashboard is proposed called Process Engine Benchmarking Interactive Dashboard (PEBDASH) which presents this information in a way such that the user can navigate to the information relevant to him. The dashboard gets its information from a database which can accumulate a variety of different benchmarks with their results to allow for comparison over time using any previously loaded data. Hence, the following hypothesis is posed:

Hypothesis H4.4 *An interactive dashboard is a suitable form to present benchmarking results and support selection decisions.*

Efficiency and Feasibility

Especially the proprietary standard-based process engines come with complex and long running installation and startup procedures. For instance, the Oracle Business Process Engine that is part of the Oracle SOA Suite 11gR1 middleware comes with an installation guide [193] that requires the user to download five files summing up to five GB and following necessary installation steps stated on 48 pages. Lenhard et al. [150] showed that installation and startup time varies greatly between open source process engines as well. This puts a burden on conducting benchmarks with such process engines, resulting in a much higher *time to result*. For larger benchmarks, this may result in waitings days and not hours for the results. To overcome this challenge, the use of virtualization is proposed as part of the Efficient Process Engine Benchmark Framework (ePEBWORK) for creating an execution efficient version of PEBWORK. The central idea is to be independent of the actual installation and startup times by using and restoring snapshots of virtual machines with a fresh and already installed as well as started process engine. Hence, the following hypothesis is proposed:

Hypothesis H4.5 *By leveraging virtualization it is possible to improve the efficiency of a benchmarking framework significantly.*

Knowledge Representation

Solving all the previously stated challenges creates a set of solutions. To prevent that other researchers and practitioners have to pose and solve these challenges over and over again that knowledge should be captured. It is proposed to capture it in the form of patterns [5]. Those patterns are named Process Engine Benchmarking Pattern Candidates (PEBPATT) in this work as they are the first iteration towards a full pattern catalog. Hence, the following hypothesis is proposed:

Hypothesis H4.6 *Patterns are a suitable form to describe the central elements of process engine benchmarking.*

1.4. Method

To support the hypotheses, design science is applied. Design science "creates and evaluates IT artifacts intended to solve identified organizational problems" [108, p. 77]. Such IT artifacts are categorized by March and Smith [164] into *constructs* (i.e., languages), *models* (i.e., problems and solutions using an existing language), *methods* (i.e., guidelines, practices, or algorithms), or *instantiations* (i.e., tools and implementations) [108, 164, 284]. Because the term *model* is used differently in a variety of contexts, in this section, the definition by March and Smith [164] that "models represent situations as problem and solution statements" [164, p. 256] is followed. Models and methods are closely related: methods deliver solutions based on specified problems whereas models represent both, problems and solutions [284]. In this work, a variety of IT artifacts is built and evaluated. Hence, design science fits perfectly as a research method. To ensure that design science is applied correctly (i.e., in conjunction with approved scientific practice), the seven guidelines of Hevner et al. [108] and their relation to the thesis will be described shortly in the following.

Design Science

Table 1.1.: Artifacts Categorized using Terminology by Hevner et al. [108]

Artifact Type	**Artifact**	**Evaluation Methods**
Construct	PEBL	Analytical (Expressiveness)
Model	PEAL	Descriptive (Scenarios)
Model	Benchmarks	Analytical (Static, Dynamic)
Method	PEBWORK	Analytical (Good Benchmark Criteria)
Method	ePEBWORK	Analytical (Performance)
Method	PEBPATT	Analytical (Challenges, Relationships)
Instantiation	PEAL Prototype	Testing (Scenarios)
Instantiation	betsy	Testing (Benchmarks), Analytical (Results)
Instantiation	vbetsy	Experimental (Performance compared to betsy)
Instantiation	PEBDASH	Analytical (Requirements), Testing

In this work, a variety of different artifacts is designed as shown in Table 1.1, ranging from constructs, models, and methods to instantiations (Guideline 1: Design as an Artifact). They solve a relevant problem, as they help to close the information gap by allowing the decision maker to obtain the necessary information (i.e., objective information about the quality of process engines) as already outlined at the beginning of this chapter (Guideline 2: Problem Relevance). The IT artifacts are evaluated using different kinds of evaluation methods as shown in Table 1.1, comprising descriptive, analytical, experimental, or testing-based evaluation methods (Guideline 3: Design Evaluation). The research contributions comprise the created IT artifacts given in Table 1.1 (Guideline 4: Research Contributions). These artifacts have been created and evaluated with the necessary research rigor, e.g., by applying formal methods,

Guidelines

peer-review within the research group, and conducting experiments thoroughly (Guideline 5: Research Rigor). This dissertation project started with initial versions of PEBL, PEBWORK, and PEAL for a conformance benchmarking of BPEL engines, and since then, these artifacts have been iterated and improved over time to cover more engines, additional quality characteristics, other process languages, and higher efficiency (Guideline 6: Design as a Search Process). Last, in Section 1.5, the research is communicated to academia primarily through a plethora of publications and to industry primarily through the open sourced prototypes, such as the dashboard[14] visualizing the produced benchmark results (Guideline 7: Communication of Research).

1.5. Contributions

The contributions of this work are 1) *concepts*, 2) *benchmarks*, 3) *prototypes*, and 4) *publications*. Each of them is outlined in the following.

Concepts

The core contributions comprise six concepts that, together, form a body of knowledge for benchmarking process engines: Process Engine Abstraction Layer (PEAL), Process Engine Benchmark Language (PEBL), Process Engine Benchmark Framework (PEBWORK), Efficient Process Engine Benchmark Framework (ePEBWORK), Process Engine Benchmarking Interactive Dashboard (PEBDASH), and Process Engine Benchmarking Pattern Candidates (PEBPATT). Each is detailed in its chapter, as stated in Section 1.6, and together, they support hypotheses H4.1 up to H4.6.

Benchmarks

As part of the evaluation of the concepts, benchmarks have been created. These benchmarks, however, are contributions by themselves, including the methods applied to come up with the benchmark. General ideas of the methods to produce good benchmarks are described (Section 4.5.1). The application of the benchmarks, however, is subdivided into BPEL-based benchmarks regarding conformance in Section 4.5.2.1, expressiveness in Section 4.5.2.2, static analysis in Section 4.5.2.3, robustness in Section 4.5.2.4, and BPMN-based benchmarks regarding conformance in Section 4.5.3.1 and expressiveness in Section 4.5.3.2. The key points of the results of these benchmarks for different BPEL and BPMN engines are available in Section 5.5.2.

Prototypes: betsy, vbetsy, and dashboard

The concepts are also evaluated regarding their feasibility through prototypes. All prototypes are open source and publicly available on GitHub. The concepts are implemented in two prototypes: *betsy* and the *dashboard*. *BPEL/BPMN Engine Test System (betsy)*[15] is the prototype for evaluating PEAL, PEBL, PEBWORK, and ePEBWORK. It supports open source BPEL and BPMN engines in different versions and configurations within local or virtual environments. It includes BPEL as well as BPMN benchmarks [81, 94–96, 98–100].

[14] `https://peace-project.github.io/`, visited 2017-3-31
[15] `https://github.com/uniba-dsg/betsy`, visited 2017-3-31

Although the variant of betsy that uses virtualization is called virtualization-enabled betsy (vbetsy), it is integrated into betsy itself [100]. The dashboard[16] is an HTML5 web application and the prototype for PEBDASH. It visualizes data from benchmarks described in PEBL [18].

Other Prototypes

Additional prototypes have been implemented as well. They are either early versions of the previously mentioned prototypes, separate prototypes that are used by betsy directly, or completely separate prototypes. *Uniform BPEL Management Layer (UBML)*[17] is a uniform Web Service and Java API for seven open source BPEL engines in different versions and configurations [97]. It is a preliminary version of the implementation of PEAL. Similarly, the *betsy-dashboard*[18] is a rudimentary HTML dashboard for betsy, which has been superseded by the actual *dashboard* prototype. The prototype betsy makes use of *BPELlint* and *BPMNviz*. *BPELlint*[19] is a static code analysis tool for BPEL that supports 71 out of the 94 static analysis rules [101]. And *BPMNviz*[20] is a software that supports automated image generation from a BPMN file. Based on *BPELlint*, an IntelliJ IDEA Plugin[21] called *BPELlint-idea* has been created to use *BPELlint* within IntelliJ IDEA. It is published to the plugin repository[22].

Publications

The contributions described in this thesis are backed by 22 publications as shown in Table 1.2. Of these 22 publications, 17 papers are peer-reviewed. Most of the peer-reviewed papers are published as part of international conferences and workshops, and two of these papers have been extended to journal publications as well. The author of the dissertation is the first author in nine of the 17 peer-reviewed publications. This dissertation is based on the research produced and already published in these 22 publications. It puts the produced research into a consistent shape, and, on top, adds additional insight and discussion as well. For each upcoming chapter, it is made explicit upon which of these publications the respective chapter is based on. Most of the research has been produced in collaboration. Only those parts that have been primarily researched by the author are included in this work.

Furthermore, apart from already published work, three papers are in the publishing pipeline that build on the contributions of this work. First, an extended pattern language [103] that builds upon the pattern candidates [102] is under review. Second and third, the lessons learned from having benchmarked a plethora of process engines is gathered as a to be published poster [63] and as a conference paper [151] under review.

[16] `https://github.com/peace-project/dashboard`, visited 2017-3-31
[17] `https://github.com/uniba-dsg/ubml`, visited 2017-3-31
[18] `https://github.com/uniba-dsg/betsy-dashboard`, visited 2017-3-31
[19] `https://github.com/uniba-dsg/BPELlint`, visited 2017-3-31
[20] `https://github.com/uniba-dsg/BPMNviz`, visited 2017-3-31
[21] `https://github.com/uniba-dsg/BPELlint-idea`, visited 2017-3-31
[22] `https://plugins.jetbrains.com/idea/plugin/7709-bpellint-idea`, visited 2017-3-31

Table 1.2.: Contributions: Publications by Type and Date

Type	Year	Where	Publications	Σ 22
Journal	2016	STT	[82]	1
	2017	FGCS	[85]	1
Conference	2012	SOCA	[150]	1
	2013	SOCA, ICSOC	[95, 96]	2
	2014	SOCA, SEKE	[98, 99, 212]	3
	2015	SOSE	[81, 101]	2
	2016	SOSE	[84, 176]	2
Workshop	2014	OTMW, ICSTW, ZEUS, CLOUDCYCLE	[93, 97, 100, 201]	4
	2016	ZEUS, PEaCE, PATTWORLD	[18, 83, 102]	3
Tech. Report	2012	BB-WIAI	[94]	1
	2014	BB-WIAI	[202, 210]	2

1.6. Outline

Three Parts

The thesis itself is structured into three parts. The problem and the relevant background is stated in Part I, followed by the proposed approach regarding process engine benchmarking in Part II which details the six core contributions along with their respective prototypes and evaluations. The thesis is concluded in Part III.

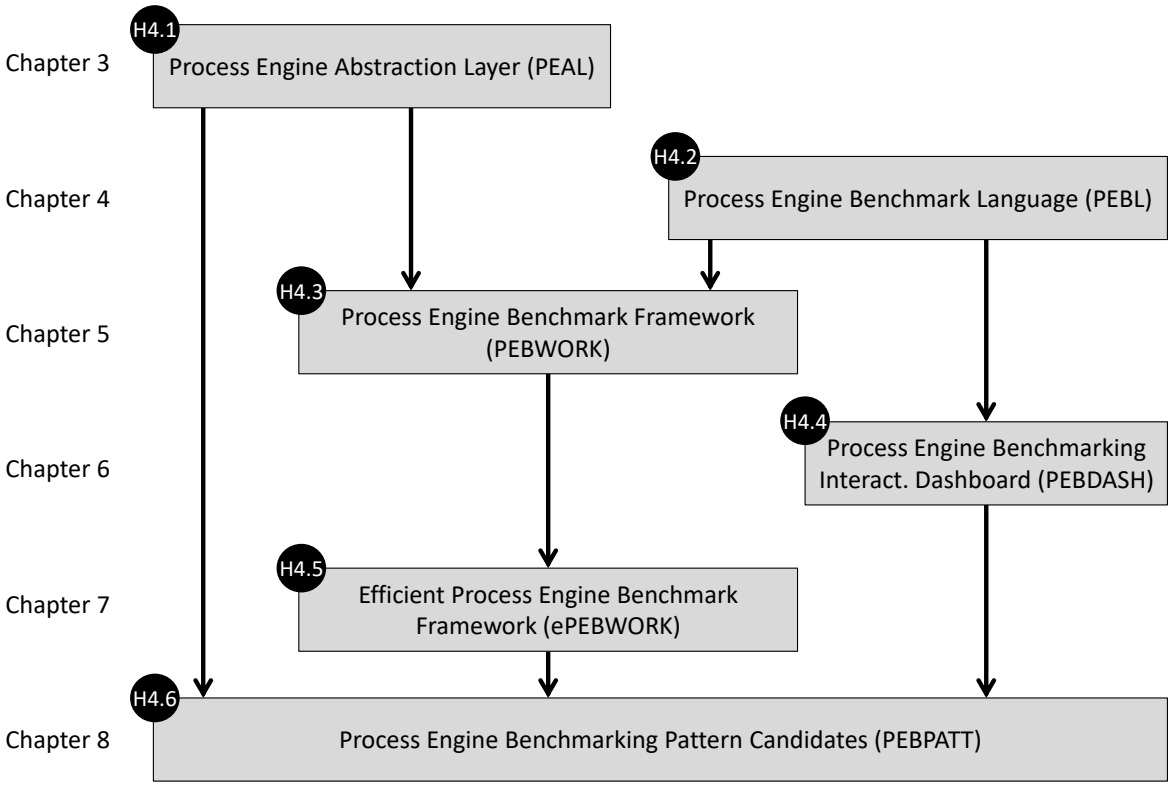


Figure 1.2.: Structure of Part II: The six core contributions and their dependencies

Nine Chapters

Part I introduces the work in Chapter 1. The background for the remainder of the work is presented in Chapter 2. Part II is structured into six chapters which build upon one another as visualized in Figure 1.2. Each chapter supports its hypothesis through a concept, prototype, and evaluation. Chapter 3 introduces the Process Engine Abstraction Layer (PEAL) and Chapter 4 defines the Process Engine Benchmark Language (PEBL). Together, they build the foundations which are necessary for the Process Engine Benchmark Framework (PEBWORK) in Chapter 5. In Chapter 6, the Process Engine Benchmarking Interactive Dashboard (PEBDASH) visualizes data encoded in PEBL. The more efficient version of PEBWORK, the Efficient Process Engine Benchmark Framework (ePEBWORK), is presented in Chapter 7. Last, the challenges and solutions that have been applied so far are captured as Process Engine Benchmarking Pattern Candidates (PEBPATT) in Chapter 8. The thesis is concluded in Part III with a summary of the major contributions, the discussion of competing approaches, and an outlook on future work in Chapter 9.

If I have seen further, it is by standing on the shoulders of giants.

Isaac Newton

2. Background

Background Overview

This work is built upon and related to established methods and terms. In this work the software tools are evaluated by comparison of their characteristics using existing decision making methods. A brief overview of decision theory and its most relevant decision making methods is given in Section 2.1. As this work focuses on process engines, the runtime environments for business processes and workflows, Section 2.2 provides an in-depth introduction into Business Process Management and its corresponding standards, languages, lifecycles, and patterns. Instead of comparing the characteristics of process engines, this work focuses on revealing the actual properties of those engines. Those properties (i.e., the actual characteristics or decision-relevant information) are obtained through benchmarking and testing. Hence, an overview of software testing, benchmarking, and software quality is given in Section 2.3. Last, another goal of this work is to provide an efficient benchmarking of process engines. Hence, Section 2.4 discusses the advantages and disadvantages of virtual environments, as well as their suitability for improving the benchmarking efficiency.

2.1. Software Selection Decisions

Software Selection Decisions

Regarding the selection of the best fitting alternative out of a set of competing alternatives, several methods have been developed over time. Section 2.1.1 outlines decision theory, the foundation for evaluating different alternatives. Most of its original methods focus on a single criterion. As the selection of a software product requires not only a single criterion but multiple criteria, the concept of Multi Criteria Decision Making (MCDM) as a more sophisticated way of decision making is described in Section 2.1.2. In Section 2.1.3, the popular MCDM method Analytic Hierarchy Process (AHP) [222] is detailed to provide a better understanding how the results produced within this work can be applied in an actual selection decision.

2.1.1. Decision Theory and Decision Making

Decision Theory & Making

Decision theory is an important discipline within the field of operations research. Overall, decisions "involve three components – acts, states, and outcomes, with the latter being ordinarily determined by the act and the state under which it takes place" [207, p. 6] and are made by the so-called *decision makers*. In other

words, "[d]ecision making is the study of identifying and choosing alternatives based on the values and preferences of the decision maker. Making a decision implies that there are choices to be considered, and in such a case we want not only to identify as many of these alternatives as possible but to choose the one that best fits with our goals, objectives, desires, values, and so on." [104] To help the decision makers to make decisions, decision tables and decision trees are proven methods [173, 246]. With these the decision maker can capture and describe the acts, states, and outcomes as well as the relationships between them. In a decision table the acts and states are the columns and rows whereas the outcomes are put in the cells depending on which act is performed in which state [207]. A decision tree is a hierarchical version of the flat decision table [207] in which each leaf represents an outcome and each edge either a state or an act leading towards the outcome.

Decision Problem

In the context of this work, the *acts* represent all the different process engines from the various vendors – each process engine being another option. The *states* are the different usage scenarios in which the engines will be used. These usage scenarios are normally captured by a list of requirements or user stories. The *outcomes* may be measured in money earned or wasted, which indirectly depends on the extent the requirements are fulfilled. The issue is that there are a great number of different states and, therefore, outcomes, as a decision of choosing the best fitting software system involves many criteria. And this plethora of criteria has to be grouped to make it more approachable (e.g., in a hierarchy). Consequently, a decision table or tree would not suffice as they cannot capture all the different criteria and their possible values and influences on the outcomes in a way useful to the decision maker. Furthermore, decision makers can either be individuals or groups of varying sizes. And the type of decision maker influences which decision making technique has to be chosen. Our targeted problem may involve either an individual IT expert making the decision in a small company or a group of both IT and business people in large ones. This is also not captured directly in a decision tree or table. Therefore, a more sophisticated method for making the decision is necessary which can be found in the concept of Multi Criteria Decision Making (MCDM).

2.1.2. Multi Criteria Decision Making

MCDM Taxonomy

An overview of the field of Multi Criteria Decision Making (MCDM) is given by Triantaphyllou [254]. According to Zimmermann and Gutsche [295], MCDM is subdivided into Multi Object Decision Making (MODM) and Multi Attribute Decision Making (MADM). They differ only in their decision space (i.e., their alternatives). For MODM, the decision space is continuous whereas it is discrete for MADM. The selection problem in this work is about already known process engines and which of them fits a specific use case best. This type of decision clearly has a discrete decision space with the different process engines being the

different countable alternatives. Hence, this work is treating a MADM problem. Since most MCDM decisions do have a discrete decision space and therefore are MADM decisions, the two terms are often treated synonymously [254]. Throughout this work, the more well known term MCDM is used for the MADM decision problem of selecting the best fitting process engine.

MCDM Methods

For solving MCDM problems, there are a plethora of methods available that have been developed over time, e.g., the Weighted Sum Model (WSM) [68, 254], Weighted Product Model (WPM) [27, 172], Analytic Hierarchy Process (AHP) [222], revised AHP [13], Analytic Network Process (ANP) [223], Elimination and Choice Expressing Reality (ELECTRE) [67, 216] and Technique for Order Performance by Similarity to Ideal Solution (TOPSIS). Every method available is based on three steps according to Triantaphyllou [254]:

> "1. Determine the relevant criteria and alternatives.
> 2. Attach numerical measures to the relative importance of the criteria and to the impacts of the alternatives on these criteria.
> 3. Process the numerical values to determine a ranking of each alternative." [254, p. 5f.]

They have two things in common: the alternatives and the decision criteria [33]. Our MCDM problem of deciding which process engine to choose fits best with AHP as most criteria are structured in a hierarchy as it is normally done in RfIs and most decisions are made in a group which is also supported by AHP. Among the many methods for multi-criteria decision making (i.e., Figueira et al. [66] describe 24 different ones), AHP also achieves the best ratings within a comparison of MCDM methods using the same data [28].

2.1.3. Analytic Hierarchy Process

AHP

The Analytic Hierarchy Process (AHP) [222] and its more abstract form Analytic Network Process (ANP) [223] from Saaty, is, in fact, one of the most widely used and popular MCDM methods. But while AHP uses a hierarchical structure for the factors of the decision, ANP uses a graph structure instead. Hence, AHP is a subset or special configuration of an ANP as one can emulate a tree structure with a graph. Since its inception in 1990 AHP has seen much use. Vaidya and Kumar [256] found 150 papers that applied AHP and analyzed 27 of them in detail. One of its uses is the selection of software products or systems. For instance, Lai et al. [142, 143] used it to choose the best multimedia authoring system whereas Wei et al. [277] determined the best Enterprise Resource Planning (ERP) system with it. Hence, AHP is well-suited for our task of evaluating process engines.

AHP in Detail

The AHP method is divided into three steps. First, the decision problem has to be structured into a tree with at least three levels. Each criterion can be subdivided into an arbitrary large subcriteria tree. For reasons of

comprehensibility the method is described here without any subcriteria using a tree with exactly three levels in the following. On the first and upmost level one puts the *goal*, on the second level one puts the factors or decision *criteria*, and on the third and lowest level one puts the different *alternatives*. In the second step, the decision criteria are compared pair-wise by stating the relative importance towards the goal on a scale from 1 up to 9. For instance, a 1 means both are equally important whereas 9 means that one is extremely important in comparison to the other. From this, we can determine the local criteria priority by computing the eigenvalues of the comparison matrix. Next, the pair-wise comparison is conducted on the lowest level by comparing the alternatives stating the relative importance towards each decision criterion. From this, we can determine the local alternative priority by computing the eigenvalues of each of the comparison matrices. Third, the global priorities are computed by summing up the multiplication of the criteria priority from level two with the alternative priority from level three to a single value per alternative. The alternative with the highest global priority is the best according to this approach [222].

Nowadays, a revised AHP is also available because of shortcomings of the original AHP which were uncovered by Belton and Gear [13]. Because these shortcomings only occur on special occasions and the revised AHP is more complicated to apply, the original AHP is still in use and recommended today, for instance by the Department of Energy, USA [8]. Revised AHP

2.2. Business Processes and Workflows

While AHP is useful for the evaluation and comparison of many kinds of software, this work focuses specifically on process engines. Process engines are the runtime environment for business processes and workflows and the heart of large Business Process Management Systems (BPMSs) [278]. A complete BPMS supports the full Business Process Management (BPM) method which specifies the lifecycle of a business process. The phases of this lifecycle are described in more detail in Section 2.2.1. The business processes themselves are described using languages which are specified in long and complicated standards as described in Section 2.2.2. Process engines implement these standards and can execute processes defined in the languages described by the standard. An overview of them is given in Appendix A. BPM

2.2.1. Business Process Management and its Lifecycle

As the name suggests, the BPM lifecycle is centered around business processes, or processes for short. One version of the lifecycle is shown in Figure 2.1. BPM comprises a four-step feedback loop to constantly improve a process (or a set of processes) [267, 278]. First, a process is designed, and then configured so that BPM Lifecycle

it can be enacted (i.e., executed). The lessons learned through the enactment of that process are captured in the diagnosis step which then leads to an improved version of that process in the next process design step. In this work the focus lies on the system configuration as in this step the user has to decide which process engine to use so that a particular process (or set of processes) is enacted according to his requirements [260].

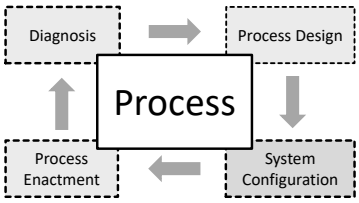


Figure 2.1.: The BPM Lifecycle, adopted from van der Aalst et al. [261, p. 5]

Workflows

In the introduction the terms revolving around BPM are defined according to Weske [278]. Besides BPM that manages business processes, there is also Workflow Management (WfM) that manages workflows. In the following these terms are compared with each other. The definition of a workflow below clearly separates a process from a workflow with the workflow being the automated part of a process. Today processes can often be executed directly on BPMSs. Hence, in that case, a workflow and a process are the same and, therefore, they are used interchangeable. In this work the terms are used in the same way as only executable processes (i.e., workflows) are covered. According to van der Aalst et al. [261], the methodology Business Process Management (BPM) extends the methodology Workflow Management (WfM) by not only concentrating on process enactment by also on diagnosis and improvement of the processes. Hence, the term BPM is used instead of WfM throughout this work.

> **Definition 2.1 (Workflow)**
> *"[A workflow is t]he automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules"* [279, p. 8]

2.2.2. Business Process Languages and Standards

Process Execution Taxonomy

The taxonomy revolving around a process and its execution is shown in Figure 2.2. The central element is the process language which consists of both the syntax (in most cases the XML elements and attributes) and semantics (i.e., execution behavior). Typically, the language is specified as part of a standard. Using the syntax of the language, models of processes can be created. Those models are typically serialized in XML. An engine executes instances of such deployed process models by following the execution semantics defined

by the standard. During the execution of the process instances messages are exchanged (received and sent) by either existing instances or new instances are created as a result. Hence, the process languages are Domain-Specific Languages (DSLs) for describing executable processes according to the following definition for a DSL.

> **Definition 2.2 (Domain-Specific Language)**
> *"A domain-specific language [...] is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."* [263, p. 26]

Process Language

Today, there are a plethora of different process languages available as shown in a comparative study by Mili et al. [171] in 2010. Since then new versions of existing process languages were created which differed greatly from their previous versions, e.g., the language BPMN contains execution semantics and an even larger feature set compared to BPMN 1.2 [188].

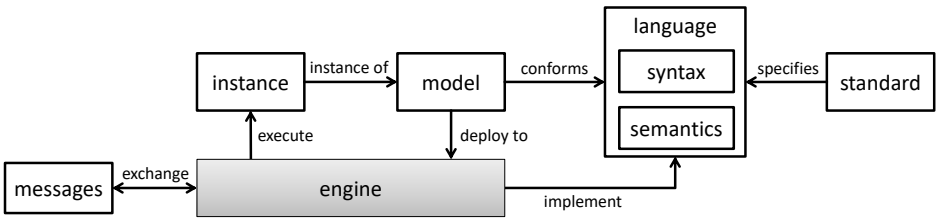


Figure 2.2.: Taxonomy: Standard, Language, Model, Instance, and Engine, partly taken from Weske [278]

Process Language Standards

Leymann et al. [155] discuss the importance of standards for BPM. The three process languages XML Process Definition Language (XPDL) 2.2 [280], Web Services Business Process Execution Language 2.0 (BPEL) 2.0 [181] and Business Process Model and Notation 2.0 (BPMN) 2.0 [115] (and their previous versions [188]) are defined through industry standards. The XPDL [280] specification has been created by the Workflow Management Coalition (WfMC), the BPEL [181] specification by the Organization for the Advancement of Structured Information Standards (OASIS) and the BPMN [189] specification by the Object Management Group (OMG) and additionally standardized by the International Standards Organization (ISO) as [115]. XPDL is not considered in this work despite being implemented in several engines [281] because XPDL is mostly only used as a serialization format and converted in either a BPEL or BPMN model for actual execution. Hence, BPEL and BPMN are detailed within this section later on.

Standard Conformance

A criterion for comparing process engines is that of language support, or which language features are supported by a process engine and which are not. Because the languages are defined in a standard, the notion of standard conformance arises. Instead of solely focusing on whether a feature is supported

by an engine, the standard conformance puts the focus on whether a language feature is supported in accordance to the standard. In this work, the notion of standard conformance is used also for the notion of language support, leaving aside any vendor-specific language features as those would prohibit portability and lead to undesired vendor lock-in. Standard conformance in regard to the language features is defined under the term feature conformance as follows.

> **Definition 2.3 (Feature Conformance)**
> *Feature conformance, or conformance, is defined as the number of supported language features of a process engine according to the standard in relation to the number of the overall language features of the language standard.*

2.2.2.1. Web Services Business Process Execution Language 2.0

BPEL 2.0

The Web Services Business Process Execution Language 2.0 (BPEL) consists of a large XML-based vocabulary to describe processes which can receive and send XML-based SOAP [270] messages in a WS-based environment that relies upon the WSDL 1.1 [271] standard. A BPEL process orders the message exchanges through its control and data flow comprising *basic* and *structured* activities as well as *scopes*.

Basic Activities

"Basic activities are those which describe elemental steps of the process behavior." [181, p. 84] Messages are exchanged through *receive*, *reply*, and *invoke* activities, and variables are read from and written to in the *assign* activity. With the *throw* and *rethrow* activities, faults are thrown and rethrown along the call hierarchy. The process instance is set on hold for a specific time with the *wait* activity or killed instantaneously with the *exit* activity. The *empty* activity simply acts as a placeholder as it has no execution semantics.

Structured Activities

The structured activities weave the control flow out of these basic building blocks. The simplest one is executing a collection of activities in *sequence*. Other typical control flow constructs known from block-structured programming languages such as a condition using the *if* activity with optional *elseif* and *else* branches as well as different loop types, for instance, the *while* and *repeatUntil* activities, are available, too. The special loop *forEach* supports the (parallel) execution of different iterations. With *pick* activities one can wait until either a message is received or a timeout is fired. Parallel execution is possible using *flow* which builds a directed acyclic graph with optional conditions for moving through the graph, providing a graph-based model. This shows that BPEL is both graph-based and block-structured [135].

Scopes

Scopes provide the execution context of activities. The *process* is the root *scope* of the XML tree. A *scope* defines *variables* and their visibility, and can have *faultHandlers*, *compensationHandlers*, and *terminationHandlers* (FCT handlers) attached which are required to handle or react upon faults or required for the teardown of the process or scope. Fault handlers can catch previously thrown

faults through *catch* and *catchAll* constructs and handle them appropriately or *rethrow* them in case they should be handled by another fault handler in the scope hierarchy. Compensation handlers can be registered to explicitly compensate previous state changes either in the process instance itself or in external systems which must be reverted or cleaned up after a fault or erroneous state occurred. The termination handlers are called when the scope is being left – they can be used to clean up external state, close any open resources, or simply to log the teardown of the scope. Another type of handler is the *eventHandler*. Such a handler can be attached to a *scope* as well. It can receive events in the form of SOAP messages as long as the scope it is attached to is active. The events arrive in parallel and the *eventHandlers* will process them in parallel, too.

Message Exchanges

A BPEL process uses the concept of message exchanges. Receiving a message creates a message exchange which is required for sending a reply to the previously received message. Especially, in case multiple different messages are received and therefore multiple different replies have to be sent to different callers.

Message Correlation

Another concept is that of correlation. When the engine receives a message, it needs to determine to which process instance this message must be routed. In BPEL one can define correlations based on the content of the message to setup such routing. Some messages can also trigger the creation of new process instances.

Static Analysis Rules

The BPEL specification comprises 94 static analysis rules [181] that are numbered from 1 up to 95, with the rule number 49 missing. These rules cover aspects that are not part of the accompanying XML Schema Definition (XSD) of the specification and "any process definition that fails one or more of these checks must be rejected by the WS-BPEL processor." [181, p. 194] Because those rules are part of the specification, they are also under the umbrella of standard conformance. To differentiate between the feature support during process enactment and the detection of errors in the configuration phase, the notion of static analysis conformance, or static analysis for short, is introduced.

> **Definition 2.4 (Static Analysis Conformance)**
> *Static analysis conformance, or static analysis, is defined as the number of rules a process engine has fully implemented to reject erroneous process models upon deployment in relation to the overall number of static analysis rules.*

CoreBPEL

The BPEL language contains a lot of syntactic sugar and can be reduced to a much smaller subset named CoreBPEL [110] which has no semantic redundancy. For instance, a *repeatUntil* can be replaced by a *while* loop using an additional temporary variable to "ensure at least one iteration and expression language independence when negating the condition" [110, p. 1988].

Standards An orchestration language like BPEL lives within a world of standards from the Web Services (WSs) ecosystem. It grounds itself on top of XML, XML Schema Definition (XSD) and the Web Service Description Language (WSDL). A plethora of WS-* standards extend the basic WSDL definition by providing Quality of Service (QoS) attributes (e.g., security [180] or reliability [182]). Moreover, there are multiple extensions available for BPEL as stated by Kopp et al. [137] (e.g., BPEL4Chor [43] and BPEL4People [4]).

2.2.2.2. Business Process Model and Notation 2.0

BPMN 2.0 The Business Process Model and Notation 2.0 (BPMN) 2.0 [115] aims to be "readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes" [115, p. 1]. Hence, the focus of BPMN is to improve Business-IT alignment by closing the gap between the design and implementation of processes [115].

Conformance The language itself can be used in four different settings and has four different conformance levels, one for each setting. It can be used solely for modeling purposes (Process Modeling Conformance), but also for execution on BPMN engines (Process Execution Conformance) according to the execution semantics [115, chapter 13] or on BPEL engines (BPEL Process Execution Conformance) by mapping BPMN processes to BPEL models [115, chapter 14]. Last, it supports the modeling of interacting processes as choreographies (Choreography Modeling Conformance) [115, chapter 11]. Using these conformance levels, one can easily a) determine the focus of a specific process model, and b) evaluate tools whether and how well they support these conformance levels. An overview of the different conformance types is also available [115, chapter 2.6].

Language Although the language allows a plethora of different kinds of processes to be defined (i.e., collaborations, conversations, choreographies, and private or public orchestrations), the focus is on the ones that can be executed according to the *Process Execution Conformance*. Hence, only the related language features available for that conformance level are discussed. A *process* is a graph of *activities*, *events*, *gateways* and *sequence flows*, the so-called *flow elements*. Each of them is detailed in the following subsequently.

Activities In the BPMN specification, there are many different types of activities. The actual work in a process model is done through *tasks* (e.g., ship product, send order, or create user). A task is an activity which can also be executed multiple times, making it a *loop activity* or *multi-instance activity* by specifying additional attributes and elements for a task. The other activities are (*ad-hoc*) *sub-processes* or calls to them via a so called *call activity*. Sub-processes are particularly useful to decompose a large process into smaller parts which can be called, creating a hierarchic structure.

Gateways route the control-flow, which may lead to concurrency, too. The standard contains *exclusive*, *inclusive*, *parallel*, *event-based* and *complex* gateways, supporting a variety of different control-flow strategies and patterns [115, p. 435]. A typical condition (i.e., an if with an else) is encoded using an exclusive gateway, whereas one can use parallel gateways to describe a fork/join pattern, creating and then merging multiple branches which execute in parallel. Event-based gateways defer the selection of different control-flow branches based on the arrival of a specific event. The first arriving event determines which control-flow branch is executed. The complex gateway allows specifying different merging behaviors through a boolean condition which handles the token semantics of BPMN directly. Gateways

Events are central to BPMN: each process starts through *start events* and ends through *end events*. In between, there are *intermediate events* or so-called *intermediate boundary events* which can be emitted and reacted upon for a variety of reasons, including the arrival of a message or the occurrence of a fault. A mechanism for compensation is also available through specific *compensation events* and corresponding handlers. Events

To structure larger processes and make them more human-readable, a *process* can be partitioned into several *lanes* of a *pool*. However, lanes are ignored by the engines as they do not have any execution semantics. Pools or participants can define their own processes. Remaining Language Elements

In BPMN, it is possible to model data flow as well. Those diagrams are used within this work to describe PEBWORK in Chapter 5. A data flow model consists of *tasks* and *data objects*. Tasks represent the functions which have inputs and outputs in the form of data objects. The actual connection of a task with a data object is implemented through a *data association*, which also specifies through its direction whether the data object is an input or output of the task. A process itself can have inputs and outputs as well. In that case, the data objects are marked with an arrow (filled for output, unfilled for input) in the upper left corner of their visual representation. A data object can also be specified as a *data store* for reuse or an *IT System* to represent different states within systems. Data Flow

2.2.3. Workflow Patterns

Another way to look at the features of process engines, in general, is that of patterns. A pattern is defined as follows by Alexander [5]. Patterns

> **Definition 2.5 (Pattern)**
> *"[A pattern] describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* [5, p. x]

Workflow Patterns

A workflow pattern is a pattern that captures specific interactions of a workflow (i.e., a pattern in the workflow domain). Within the workflow domain a plethora of pattern catalogs has been created as shown in Table 2.1. The pattern catalogs range from low-level pattern catalogs like control-flow, data, time, and service, to high-level catalogs with activity or process viewing patterns. Most of them have been created as part of the Workflow Pattern Initiative [291].

Table 2.1.: Workflow Pattern Catalogs

Pattern Catalog	Publications
Control-Flow Patterns	[257]
Service Interaction Patterns	[9, 10, 174, 259]
Time Patterns	[144]
Data Patterns	[217, 220]
Exception Patterns	[221]
Resource Patterns	[218, 219]
Activity Patterns	[250]
Change Patterns	[275, 276]
Process Viewing Patterns	[228]
Protocol Patterns	[264]
Correlation Patterns	[11]

Expressive Power

According to Felleisen [59], the expressive power (i.e., expressiveness) of programming languages is determined by "whether a programming language can express a programming construct." [59, p. 36] This refers to the effort that is attached if one wants to express a particular construct in another language. If the other language does not support this construct, and requires the use of a variety of other constructs instead as a workaround, the other language is denoted as less expressive.

Expressiveness

The workflow patterns build upon the idea of expressive power of programming languages but apply them to workflow and process languages. With these patterns, one can describe how directly a pattern can be expressed in a given language (i.e., language expressiveness). A pattern is directly supported by a given language if there is a single language construct which corresponds to the pattern at hand, and only partially supported if there is a workaround using a few language constructs together to implement the pattern (or part of it). If the workaround would require a huge amount of language constructs, the pattern is not directly supported. There are already evaluations of process languages and engines regarding the support for these patterns, but they are published for older versions of the process engines and hard to reproduce [149, 285, 288]. In the following, the term expressiveness is defined in the context of process engines.

Definition 2.6 (Engine Expressiveness)
The engine expressiveness (i.e., expressiveness of a process engine) is defined as the workflow pattern support of the process engine in relation to the workflow pattern support of the process language the process engine supports.

Table 2.2.: Language Support for Workflow Control-Flow Patterns

Control-Flow Patterns[257]		**BPEL**	**BPMN**
WCP-01	Sequence	+	+
WCP-02	Parallel Split	+	+
WCP-03	Synchronization	+	+
WCP-04	Exclusive Choice	+	+
WCP-05	Simple Merge	+	+
WCP-06	Multi-Choice	+	+
WCP-07	Structured Synchronizing Merge	+	+/-
WCP-08	Multi Merge	-	+
WCP-09	Structured Discriminator	-	+/-
WCP-10	Arbitrary Cycles	-	+
WCP-11	Implicit Termination	+	+
WCP-12	MI Without Synchronization	+	+
WCP-13	MI With A Priori Design-Time Knowledge	+	+
WCP-14	MI With A Priori Run-Time Knowledge	+	+
WCP-15	MI Without A Priori Run-Time Knowledge	+	-
WCP-16	Deferred Choice	+	+
WCP-17	Interleaved Parallel Routing	+/-	+/-
WCP-18	Milestone	+/-	-
WCP-19	Cancel Activity	+/-	+
WCP-20	Cancel Case	+	+

Language Expressiveness

In Table 2.2, the pattern support is provided for the two process languages (i.e., their language expressiveness). It reveals that the two languages do not have the same expressiveness: BPMN, actually, is more expressive than BPEL. This is mainly because BPEL cannot directly handle the patterns WCP08, WCP09, and WCP10.

Criticism

Despite the huge popularity of the workflow patterns, there is also criticism. According to Börger [24], the workflow patterns fail to "provide practitioners with a suitable means [to] precisely and faithfully [...] capture business scenarios and to analyze, communicate and manage the resulting models" [24, p. 305]. Nevertheless, for this work, the workflow patterns still can act as a helpful comparison criterion stating how well an engine supports these patterns and how far away it is from the comparison baseline which is the support of the process language itself (i.e., engine expressiveness).

2.3. Software Evaluation

In this work, the quality of process engines is evaluated which builds upon the quality standard family of the ISO/IEC called Systems and software Quality Requirements and Evaluation (SQuaRE) described in Section 2.3.1. The method to reveal the quality of such process engines is benchmarking as described in Section 2.3.2. Benchmarking makes use of software testing which is outlined in Section 2.3.3.

2.3.1. ISO/IEC Standard Family 25000 SQuaRE

The International Standards Organization (ISO) and International Electrotechnical Commission (IEC) standardization bodies have consolidated and harmonized several previous software quality standards in a family of standards for Systems and software Quality Requirements and Evaluation (SQuaRE). Those standards are subdivided into six different divisions listed in Table 2.3. In the following, the three divisions that are important for this work are outlined: 25010 [113], 25041 [114], and 25051 [117].

Table 2.3.: The ISO/IEC 250xx standards family [113, 114, 116, 117]

ISO/IEC	Quality Division
2500n	Management [116]
2501n	Model [113]
2502n	Metrics
2503n	Requirements
2504n	Evaluation [114]
25050–25099	Extension, including RUSP Evaluation [117]

2.3.1.1. ISO/IEC 25010

Product Quality Model

The ISO/IEC 25010 standard defines a quality in use model and a product quality model. We solely focus on the product quality model that puts the product into the foreground in contrast to the quality in use model that concentrates on the perceived use of the product in a working environment. The product quality model is hierarchically structured: it has characteristics (which can be subdivided into subcharacteristics, sub-subcharacteristics, and so on) which are described through quality properties. The product quality model measures the target system and is influenced by the target computer system (includes computer hardware, non-target software, target data and non-target data). It categorizes quality into eight characteristics illustrated in Figure 2.3: functional suitability, performance, compatibility, usability, reliability, security, maintainability, and portability. “The models can, for example, be used by

[...] acquirers [...], particularly those responsible for specifying and evaluating software product quality." [113, p. 1] The standard also covers properties and measures. Measures can reveal properties that are then categorized using the quality characteristics. In the following both are detailed in more depth.

> **Definition 2.7 (Quality Model)**
> *"[A quality model is a] defined set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality."* [113, p. 19]

Properties

The standard defines software properties as either assigned or inherent properties. Assigned properties are "[m]anagerial properties like for example price, delivery date, product future, product supplier" [113, p. 30]. They can be "changed without changing the software" [113, p. 30]. Inherent properties, in contrast, are properties that are caused by the software source code and its inherent behavior. They comprise the domain-specific functional properties and quality properties. In this work the focus lies on inherent properties. Assigned properties are out of scope. In the ISO standard an attribute is an inherent property (see definition below).

> **Definition 2.8 (Attribute)**
> *"[An attribute is an] inherent property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means."* [113, p. 18]

Measures

In addition, there are internal and external measures. We focus on external measures. "External measures of system/software quality provide a "black box" view of the system/software and address properties related to the execution of the software on computer hardware and an operating system" [113, p. 31].

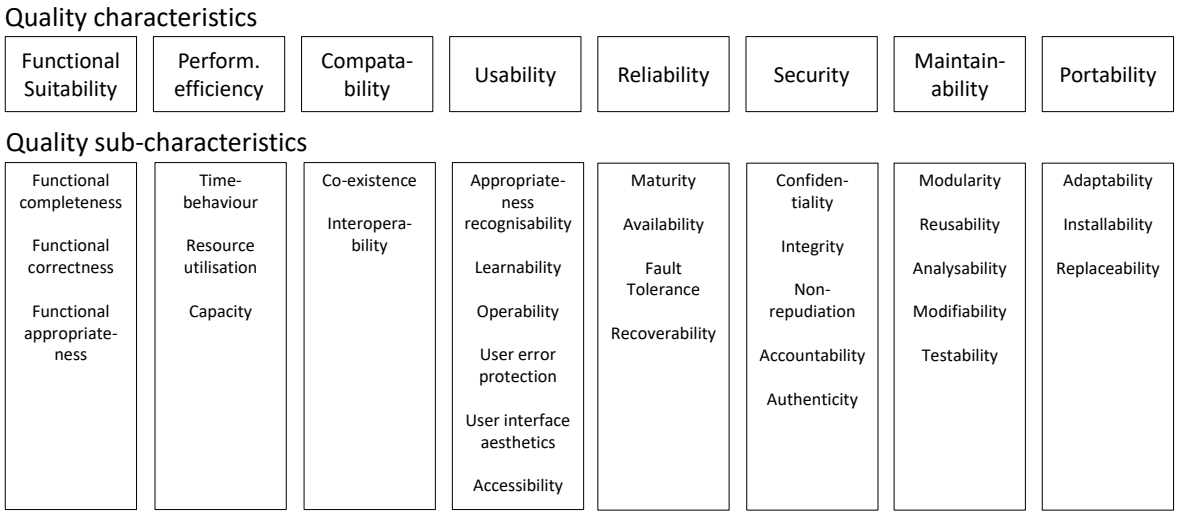


Figure 2.3.: Product Quality Model of the ISO/IEC 25010 standard [113]

They can be measured by testing dynamic properties. The external measure of software quality is defined in the standard as follows.

> **Definition 2.9 (External Measure of Software Quality)**
> *"[An external measure of software quality is a] measure of the degree to which a software product enables the behaviour of a system to satisfy stated and implied needs for the system including the software to be used under specified conditions."* [113, p. 16]

2.3.1.2. ISO/IEC 25041

ISO/IEC 25041

The ISO/IEC 25041 standard [114] is an evaluation guide for developers, acquirers, and independent evaluators of software. Hence, it is a good fit for this work. The standard defines evaluation as follows: "Evaluation is the systematic determination of the extent to which an entity meets its specified criteria. The evaluation of product quality is vital to both the acquisition and development of software. The relative importance of the various characteristics of software quality depends on the intended usage or objectives of the system of which the software is a part; products need to be evaluated to decide whether relevant quality characteristics meet the requirements of the system." [114, p. vi] The standard contains an evaluation procedure to evaluate RUSPs [114, p. 31]. This evaluation procedure can be combined with a MCDM method such as AHP to make a decision which RUSP should be selected. In that regard it is similar to the RUSP selection by Lawlis et al. [145] and Tarawneh et al. [247].

Links to Thesis

In this work, the goal is to provide information about the quality of the dynamic product ("system or software product that is measurable during execution in testing and/or operational environment" [114, p. 2]). Hence, it helps in several steps of the norm's evaluation procedure: step 4 ("[e]valuate software products based on external evaluation results, product documentation, product operating experience, product prototyping, other product evaluation methods" [114, p. 31]), step 6 ("[p]erform acceptance testing and accept/reject the product" [114, p. 31]), and possibly step 7 ("[p]erform any additional evaluation" [114, p. 31]). The PEBWORK framework and the betsy prototype can be seen as an evaluation tool ("instrument that can be used during evaluation to collect data, to perform interpretation of data or to automate part of the evaluation" [114, p. 2]).

2.3.1.3. ISO/IEC 25051

Link to Thesis

The ISO/IEC 25051 standard [117] provides a set of requirements for evaluating RUSPs and requirements for testing RUSPs against its requirements [117]. The test documentation comprises a test plan, test descriptions and test results. Those terms are similar to the ones described in the test taxonomy later in

Section 2.3.3.1. Hence, we use the terms from the test taxonomy instead of those defined in that standard.

2.3.2. Benchmarking

Definition

Benchmarking is a way to reveal properties of software products. In the domain of computing a benchmark is defined by the OED as a "program or set of programs used as a standard against which the performance of other programs (or the computer systems running them) is compared or evaluated" [186]. The verb "benchmarking" in this case is the "measurement of a computer system's capabilities, using standardized or agreed performance-testing programs, whose results may be used for comparison with those of other computers" [186]. Both definitions have in common that they focus on performance evaluations. The definitions by the Institute of Electrical and Electronics Engineers (IEEE) do have a different point of view. For them, a benchmark is known as a "standard against which measurements or comparisons can be made" [203, p. 12] as well as a "procedure, problem, or test that can be used to compare systems or components to each other or to a standard" [203, p. 12]. In that case benchmarking is seen as a more general means to compare systems using various criteria not only focusing on performance alone, which fits the goal of this work. Hence, the latter definition is used throughout this work.

> **Definition 2.10 (Benchmarking)**
> *"[Benchmarking is a] procedure, problem, or test that can be used to compare systems or components to each other or to a standard."* [203, p. 12]

Organizations

In the past, both the Standard Performance Evaluation Corporation (SPEC) and the Transaction Processing Performance Council (TPC) have created numerous benchmarks focusing historically first on hardware (CPUs) and relational databases (SQL). Later they created more and more benchmarks for different types of systems and software (e.g., virtualization using the SPEC VIRT [243] or the TPC-VMS [253]). In 2009, the SPEC created a committee to develop a Service-Oriented Architecture (SOA) benchmark[23] named SPEC SOA. It was planned to cover a whole SOA including a BPMS using BPEL processes but the subcommittee has been dissolved and no benchmark has been published.

Good Benchmarks Huppler [111]

Huppler [111] defines five characteristics of a benchmark which are shown in Table 2.4. For him a good benchmark is relevant, repeatable, fair, verifiable and economical. However, Huppler considers them disparate as a trade off between a benchmark being relevant versus being repeatable, fair, verifiable, and economical. A relevant benchmark can produce results which help to achieve some value (e.g., to make better decisions). Hence, relevance is the

[23] `https://www.spec.org/soa/`, visited 2017-3-31

Table 2.4.: Five Characteristics of a Benchmark by Huppler [111, p. 19]

Characteristic	Description
Relevant	A reader of the result believes the benchmark reflects something important.
Repeatable	There is confidence that the benchmark can be run a second time with the same result.
Fair	All systems and/or software being compared can participate equally.
Verifiable	There is confidence that the documented result is real.
Economical	The test sponsors can afford to run the benchmark.

primary characteristic as it influences the value of the benchmark directly. The other characteristics can be seen as secondary characteristics as they can influence how certain (i.e., verifiable), comparable (i.e., fair), reproducible (i.e., repeatable), and costly (i.e., economical) the benchmark is and consequently the produced results.

Table 2.5.: Seven Characteristics of a Benchmark by Sim et al. [229, p. 6]

Characteristic	Description
Accessibility	easy to obtain and easy to use
Affordability	efficient cost-benefit ratio
Clarity	clear, self-contained and as short possible benchmark specification
Relevance	representative benchmarked task
Solvability	solving the benchmarked task should be doable
Portability	benchmark should work on different platforms, architectures, etc.
Scalability	work with tools or techniques at different levels of maturity

Good Benchmarks Sim et al. [229]

According to Sim et al. [229], the "creation and widespread use of a benchmark within a research area is frequently accompanied by rapid technical progress and community building" [229, p. 1]. They state that this is because a benchmark motivates the comparison of different products, provides example tasks how to use the products, and measures the fitness of a product for its purpose. This stands in contrast to a case study or an example as both lack at least one of these criteria. Sim et al. [229] deduced seven characteristics for a successful benchmark that are shown in Table 2.5. For them, a good benchmark should be *accessible*, *affordable*, *clear*, *relevant*, *solvable*, *portable*, and *scalable*.

Good Benchmarks Summary

These seven characteristics presented by Sim et al. [229] are similar to the five ones from Huppler [111] to some extent. The mapping is shown in Table 2.6. Both focus on different aspects, and can be combined for a better list comprising nine characteristics of a good benchmark.

Evaluation

The combined characteristics will be used in this work to evaluate the benchmark language PEBL and the benchmarks themselves in Chapter 4 to show that the language is suited for encoding good benchmarks. Furthermore, the

Table 2.6.: Comparison of the Characteristics for Good Benchmarks by Huppler [111] and Sim et al. [229]

Sim et al. [229]	Huppler [111]	Combined Characteristics
Affordability	Economical	Affordability
Relevance	Relevant	Relevance
Portability	Fair	Portability
Accessibility		Accessibility
Clarity		Clarity
Solvability		Solvability
Scalability		Scalability
	Repeatable	Repeatable
	Verifiable	Verifiable

benchmarking framework PEBWORK is also evaluated with those criteria in Chapter 5 to show that it helps in conducting good benchmarks, guaranteeing important characteristics such as repeatability for free. Hence, the developer can focus on defining what the benchmark should measure and leave the rest to the framework.

Other Benchmarks

In industry, benchmarking is widely applied by comparing the processes, products or strategies with the ones of the market leader to imitate them [51]. Of these only the product benchmark is related to this work – as a company may compare their BPM engine with the ones of the market leader to imitate the features and characteristics of the probably better product. Drew [51] summarizes the challenges for doing benchmarking as follows: "Benchmarking can be both expensive and time-consuming. Ways must be found for doing it faster, more effectively and economically, without sacrificing rigour or integrity of approach." [51, p. 439]

2.3.3. Software Testing

Software Testing

A way of benchmarking a software system is to use software testing. A rather informal definition of the term software testing is provided by Myers: "Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended." [175, p. 8] The IEEE standard glossary is more formal, stating that testing is "[t]he process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspects of the system or component" [203, p. 76]. The definition of software testing used in this work is based on the understanding of the IEEE glossary, and reads as follows:

> **Definition 2.11 (Software Testing)**
> *"[Software testing] evaluate[s] the features of [...] software items."* [203, p. 76]

This work builds upon this definition of software testing to determine which features a process engine has and also to detect possible bugs along the way if features are not working as expected. In the following, all used terms and methods related to software testing will be detailed.

2.3.3.1. Test Taxonomy

Test Taxonomy

Establishing a basic taxonomy, *testing* means that a *test driver* executes one or more *test case specifications* using a *test procedure* within a *testbed*, effectively testing the system under test, to produce a *test report* containing the results of the test along with a *test log* and a record of all relevant events during the testing. Each test case specification describes the inputs and expected outputs as well as any execution condition to test the system under test with. The triplet of test driver, test procedure, and testbed executes the test, taking into account the test case specification. The testbed represents the environment which makes the test possible, including the hardware, and all required software tools and systems. The test procedure is an algorithm for testing, which is executed by the test driver [203].

Taxonomy Mapping

In this work, the test case specification corresponds to a PEBL serialization of a set of tests and a set of engines (i.e., system under test). PEBWORK is the test driver that executes such a test case specification by performing the test procedure as well as setting up and tearing down the test bed when necessary. The test report and the test log are part of the resulting PEBL serialization of the test result.

2.3.3.2. Test Design

Test Strategies

With the test taxonomy determined, the next step is to create the actual tests. *Test design* is all about how to come up with the test cases (i.e., which information to use to create the test cases). In general, there are two different test strategies, namely, *black-box testing* and *white-box testing*, stating what knowledge about the system under test is available. In the black-box testing strategy, one has *no* knowledge of the internals of the system under test, only its specification. In this work the tested process engines are considered black boxes although some engines are open source and their internals thus available. Since we want to be able to test any engine as long as it supports specific process languages, this information cannot be leveraged. Instead, this work solely relies on interacting with the engines through their Application Programming Interfaces (APIs) (e.g., installing/uninstalling, starting/stopping, or deploying/undeploying), and observing their state changes through the responses of

their APIs or logs. The logs are particularly important as they allow a peek inside the internal procedures, errors, and events. In contrast, in the white-box testing strategy, one *has* knowledge of the internals of the system under test. In this work, BPEL and BPMN processes of which their internal structure is known are tested to deduct whether the engine executes them as expected by their specification. Hence, this means that the white-box testing strategy is applied regarding the process models although they are primarily used as resources for the black-box testing strategy. The problem with both strategies, and testing in general, is, that the whole system cannot be covered fully because of test case explosion [175].

Test Methods

To cover the most important parts of the system there are several methods for both of the strategies [175, p. 35]. For black-box testing the techniques revolve around creating or deriving the most effective test cases from a specification. The typical methods are *equivalence partitioning* and *boundary-value analysis*, both providing a minimal amount of test cases which are representative of a larger set of the input domain through equivalence classes. There are also several other methods to derive test cases. Most rely on creating a model from which they can be generated (i.e., model-based testing). In this work, model-based testing is used to derive faulty test cases to test the static analysis rules of BPEL in Section 4.5.2.3 using a formal model of those static analysis rules. Other testing methods exist as well (e.g., combinatorial testing [141]). However, they are not used in this work. For white-box testing, the techniques revolve around creating or deriving test cases using a coverage metric related to internals of the system under test. For instance, the branch coverage metric determines whether the tests cover all branches based on the conditionals in the code. This is applied in the benchmark design so that the language features such as if or exclusive gateways are fully covered for all possible branches.

Mutation Testing

In recent years, the usage of mutation testing has spread [120]. The idea of this method is to verify the quality of the test suite by checking whether an injected error (i.e., a mutation) in the system under test is found (i.e., killed) by the test suite [187]. Mutations are created through mutation operators, which modify only a small part of the program and such mutation operators can be combined to create combinations of mutations. Usually, the mutation operators are automated in tools. These techniques, however, can also be used to derive test cases by applying mutation operators upon correct XML messages [70, 146]. We make use of this technique to create faulty XML messages from correct ones to determine the fault tolerance of the processes and their engines in Section 4.5.2.4.

2.3.3.3. Test Execution

Test Execution

After one has designed tests, they are typically executed. Determined by the *test procedure*, the execution of a test case is typically subdivided into a *setup*, *test*,

and *teardown* phase. In the setup phase the testbed is created and configured so that the test itself can be conducted in the next step. In the test phase the actual test is executed and one or more assertions are evaluated to determine the result state of the test, producing the test result of the corresponding test case specification. In the teardown phase the testbed is cleaned up so that resources which were previously acquired are free again. An important property of a test is its *test repeatability* which "indicat[es] that the same results are produced each time the test is conducted" [203, p. 76]. Making a test repeatable (i.e., reproducible) requires that all influencing factors for each test are known and replayed so that each test behaves deterministically. This has the benefit that one test cannot affect the results of the other tests, as there are no side effects then as well (i.e., test isolation). In this work the focus lies on producing repeatable results. Hence, test repeatability is crucial for this work as well.

> **Definition 2.12 (Test Repeatability)**
> *"[Test repeatability] indicat[es] that the same results are produced each time the test is conducted."* [203, p. 76]

Link to Thesis

Test repeatability is not easily achieved for large software systems such as process engines. It is often not known how the environment influences the behavior of such large software systems, and the internal state is hard to save and restore as well. Hence, this is a challenge for this work that is solved by PEBWORK and ePEBWORK in Chapter 5 and Chapter 7, respectively.

2.4. Virtualization

Virtualization

Software benchmarking and "[s]oftware testing [...] has much to gain from virtualization" [266, p. 25]. Since its inception several decades ago virtualization has become ubiquitous in the IT world of today [200]. Commodity processors implement virtualization instructions natively (e.g., Intel-VT[24] and AMD-V[25]) and several companies such as Oracle and VMware have released various virtualization software solutions. With these foundations, companies can leverage the advantages of virtualization (i.e., isolation, hardware independence, availability, and security) [293]. Cloud providers, for instance, make heavy use of these technologies and offer different service models, namely, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) solutions [169]. All these offerings have in common that they provide virtual resources on the basis of physical ones through an abstraction layer above the actual physical hardware [169]. The key technologies that enable such cloud offerings are *Virtual Machines (VMs)* and *containers*. Their functioning,

[24] `http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html`, visited 2017-3-31

[25] `http://www.amd.com/br/products/technologies/virtualization/`, visited 2017-3-31

strengths, and weaknesses are described in Section 2.4.1 and Section 2.4.2, respectively. The section is concluded with a comparison of the two virtualized environments and a non-virtualized environment, often called “bare-metal”, in Section 2.4.3.

2.4.1. VM-based Virtualization

Taxonomy

Virtual Machines (VMs) are environments for software which imitate real hardware. VMs are either *System Virtual Machine (SVM)* or *Process Virtual Machines (PVMs)* [39]. The term process refers to an OS-level process which runs within an OS and has nothing to do with a business process or process model. The SVM provides the full system environment in which an OS can run. In contrast, a PVM is the runtime of an application and its execution is tied to the execution of the application. For instance, the Java Virtual Machine (JVM) is such a PVM that is started when running an application written in the Java programming language and exits when the application terminates. In this work, the existing software (i.e., the process engine) is run in an isolated environment. Hence, a SVM is required setting the PVM out of scope [39, 239, 240].

Hypervisor

The layer between the actual physical hardware and the environment for the VMs is called *hypervisor* or Virtual Machine Monitor (VMM). The hypervisor has full access to the hardware resources and distributes them among the VMs. It also manages the lifecycle of VMs. Two types of hypervisors exist: the type 1 or native hypervisor and the type 2 or hosted hypervisor. The first one runs directly on the bare metal (i.e., on the physical hardware) whereas the second one is running within an existing host and its OS [200].

Pros & Cons

VMs offer several advantages. They do provide an isolated system environment that is hardware independent and can run on different hosts. A crash of one VM will not affect any of the others. Because of these advantages it is argued that one should run everything in VMs [32]. These advantages, however, come at a cost: a major overhead in computing, disk, memory, and network IO. Such performance costs originate from running the hypervisor on the host and an additional OS per VM. Especially the IO performance overhead has prevented the use of VMs in high-performance computing [293]. Another issue is that a VM takes a long time to start up as it has to boot a full OS. Some disadvantages can be mitigated using new concepts. The need for large amounts of disk space can be reduced by using deduplication [121] and the startup time can be reduced by taking and restoring memory snapshots of already running VM instances. Some products even offer live migration of VMs, that is, migrating a running VM from one host to another with only minimal effects on availability [35].

Hypervisor Software

Today, there are multiple companies offering hypervisors of type 2, e.g., Oracle VM VirtualBox[26], VMware Fusion[27], Parallels Desktop[28] and Microsoft Windows Virtual PC[29], and type 1, e.g., Xen[30], VMware vSphere[31] and Microsoft Hyper-V[32]. In this work, solely the type 2 hypervisor Oracle VM VirtualBox is used because this software is open source, free of charge, can be used either through a GUI or an API, and supports taking and restoring memory snapshots.

2.4.2. Container-based Virtualization

Container

Containers are a lightweight virtualization technique compared to VMs as containers run within the host OS but are isolated from the host OS and other containers. This allows for a much faster startup as only the isolation environment has to be started, not another OS. Hence, containers have an almost insignificant runtime overhead [128, 293].

Technical Foundations

Today, containers are available on both Linux and Windows. Microsoft, however, only supports containers since their Windows Server 2016 whereas Linux already supports containers since 2008. Consequently, in the following, the term containers refers to Linux containers only. Standardization is in progress for containers by the Open Container Initiative (OCI)[33]. The technical basis for containers are namespaces which were introduced with Linux kernel 2.6.24. A container is a process with its user space that runs on a Linux host and uses resources from the kernel space, e.g., RAM and the disk. Multiple containers can run on a host as it is the same as running multiple processes. The only difference is that they are isolated from each other. To achieve this, both kernel namespaces and control groups (cgroups) are used. With kernel namespaces each container gets its own environment with its network devices, mount points, CPUs, and RAM. It has no knowledge about any other container running on the same host. The cgroups limit the actual resources of the container, e.g., the amount of RAM or the number of cores [293].

Resource Overhead

As shown by Felter et al. [60], container-based virtualization uses much fewer resources and performs faster than using VMs instead. Nevertheless, Felter et al. [60] state that IO heavy applications have to be fine-tuned nevertheless, as the bottleneck is the hard disk. Similar results are given by Xavier et al. [293] as well.

[26] `https://www.virtualbox.org/`, visited 2017-3-31
[27] `http://www.vmware.com/de/products/fusion/features.html`, visited 2017-3-31
[28] `http://www.parallels.com/`, visited 2017-3-31
[29] `https://www.microsoft.com/download/details.aspx?id=3702`, visited 2017-3-31
[30] `http://xenserver.org/`, visited 2017-3-31
[31] `https://www.vmware.com/de/products/vsphere`, visited 2017-3-31
[32] `https://technet.microsoft.com/library/hh831531.aspx`, visited 2017-3-31
[33] `https://www.opencontainers.org/`, visited 2017-3-31

Docker

Although there are several products[34] supporting container-based virtualization available today (e.g., LXC[35], OpenVZ[36], or Docker[37]), Docker is the most popular one and the de facto standard. Docker Engine is the core of Docker as it allows running Docker containers. Such containers can either be created manually step by step or through a Dockerfile which automates this task. The Docker ecosystem has been growing over the last few years. This has led to additional Docker services and applications, including Docker Compose to manage applications spanning multiple containers, Docker Machine to run containers, Docker Swarm to manage clusters of containers remotely, and Docker Hub which is the public Docker Registry for storing both Dockerfiles and Images. Being a CLI tool originally, there is a GUI application called Kitematic available as well. Docker makes a distinction between containers and images. Only containers are executable but they can be saved to an image for reuse.

Reproducible Research

An explicit advantage of "the popular emerging technology Docker [is that it] combines several areas from systems research – such as OS virtualization, cross-platform portability, modular re-usable elements, versioning, and a 'DevOps' philosophy" [22, p. 1]. Hence, Docker enables reproducible research, making it possible that the computational experiment conducted by one scientist can be easily validated by another scientist who has now the means to redo the experiment with a neglectable effort. Of course, reproducing research was possible before. The effort to do this was, however, generally high. Some other approaches that try to reduce the effort to reproduce research are based on VMs and workflow systems. Docker, however, addresses the issues of reproducible research much better than these two technologies [22], lowering the barrier further [31]. Apart from reproducing research it is used in other domains as well (e.g., reproducing exploits [40]).

Faster Tests

Another important area of application for containers in general and Docker in particular is that of efficient automated testing. This is, of course, fully within the scope of this work as we strive to efficiently reveal process engine characteristics by tests. Rahman et al. [205] used Docker to speed up large test suites by splitting them into test groups and executing each group within a container in parallel.

2.4.3. Environment Comparison

Virtual Environment Comparison

A summary of the virtualization techniques is shown in Table 2.7. As can be seen, they provide different degrees of isolation. The higher the isolation, the higher is the associated overhead. Using bare metal provides no isolation but also no overhead whereas VMs provide the highest degree of isolation

[34] `https://en.wikipedia.org/wiki/Operating-system-level_virtualization#Implementations`, visited 2017-3-31

[35] `https://linuxcontainers.org/`, visited 2017-3-31

[36] `https://openvz.org/`, visited 2017-3-31

[37] `https://www.docker.com`, visited 2017-3-31

by providing a separate OS but have high memory, disk space, and startup time, as well as a medium network time overhead. This comparison is similar to the one by Rahman et al. [205] which compares containers with VMs. In our case, however, we also compare the bare metal environment and use a partly different set of comparison criteria. In addition to the general disk space, startup, and teardown overhead, we also compare both network and memory overhead as well as their capabilities to create a snapshot of memory and disk.

Table 2.7.: Virtualization Techniques

	bare metal	**VM**	**container**
isolation	none	OS	container
snapshot			
memory	(x)	x	WIP
disk	x	x	x
space overhead			
disk space	none	high	medium
+ *deduplication*	*none*	*medium*	*very low*
memory	none	high	low
time overhead			
startup time	none	high	low
teardown time	none	high	low
network time	none	medium	low

Snapshots

Snapshots are an important topic for this thesis as process engines can have long installation and startup times. In the bare metal approach, a disk snapshot could be taken by simply copying the folder the process engine is installed in. As files may be installed in other places as well during the installation, this simple method is often not sufficient. Moreover, some installations may set absolute paths making the previous installation unusable in other paths. The memory snapshot, however, is much more suited for process engines with long installation and startup times as it allows to simply restore the virtual environment to a point in time which provides an already started process engine. But this is only possible in practice for VMs. Docker is working on support for memory snapshots with experimental support in Docker 1.13 released in January 2017. The possibilities for memory snapshots of OS-level processes on a bare metal setup, however, are very limited. One example for such a system is the research prototype Berkely Lab Checkpoint/Restart (BLCR) for Linux [91]. The overhead of both, VMs and containers is studied by Agarwal et al. [3] by comparing Kernel-based VMs (KVMs) with Linux containers. The results are in line with the one in Table 2.7. With deduplication, one can reduce the disk space overhead for VMs and containers significantly. For VMs, Jin and Miller [121] outline an approach to save spaces for VM disk images by sharing

Overhead & Deduplication

blocks of data instead of using overlays (i.e., only saving the difference to an existing image). For containers, Docker uses a layered File System in which each image is immutable and based on another image, only the container which is based on an image is mutable. Hence, all images are cached and containers use the cached and immutable version of this image, resulting in exactly one copy of the image available.

Summary

In this work VMs are used to ensure test isolation and automation. Moreover, with the memory snapshots only fully available for VMs, the time to provide and destroy a fresh process engine is reduced. The container technology is simply too premature at this point in time to build an efficient benchmarking solution with it. In the future this will change and one can also benefit from the lower overhead associated with containers.

Part II.

Process Engine Benchmarking

> Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem.
>
> *David Wheeler*

3. Process Engine Abstraction Layer

Parts of this chapter have been taken from [81, 84, 94, 97, 100, 150].

In this chapter, hypothesis H4.1 *("A uniform interface is a suitable solution to interact with widely different process engines in a similar way.")* is supported.

This chapter presents the Process Engine Abstraction Layer (PEAL), which is an abstraction layer to interact with process engines supporting different process languages in a uniform way through an Application Programming Interface (API). It is used as part of the Process Engine Benchmark Framework (PEBWORK) in Chapter 5 to interact with the process engines conducting benchmarks.

3.1. Motivation

For the standardized process languages BPEL and BPMN, the key selling point was and remains portability, and with it the avoidance of a vendor lock-in [126]. Both imply that the paradigm *model once, run anywhere* holds for the process world: a process model described with a standardized process language can be executed on any process engine which conforms to the process language of the process model. It is similar to the paradigm *write once, run anywhere* in the Java world: a written Java program can be executed on any compatible JVM.

model once, run anywhere

At first glance, this appears to be sufficient. In reality, however, it is not. Figure 3.1 depicts the BPM lifecycle [261] with its four phases: *process design*, *system configuration*, *process enactment*, and *diagnosis*. Shown on the left is the status quo, with only the process design phase being engine-independent as it is based on the standardized process languages such as BPEL [181] and BPMN [115]. But even the concept of having standards is not enough to ensure portability [148]. For the other three phases, however, no standards exist. Therefore, each engine has its way of how it must be interacted with. Hence, these three phases are all engine-dependent. For instance, in the system configuration phase, BPEL as well as BPMN engines do not accept the standard-based process model for deployment but require engine-specific deployment packages containing additional deployment descriptors or other metadata

lack of standards

for which no universal standard exists [94, 150]. What is more, the actual deployment with its engine-specific deployment package requires the user to call the engine-specific deployment routines. Those deployment routines, however, also vary greatly from engine to engine, and even between different versions of an engine [94, 150]. But a successful deployment is obviously necessary to execute instances of the process in the process enactment phase. And last, the diagnosis phase, which requires analyzing (audit) logs or monitoring data lacks a standard, too.

specify once, evaluate any

In the field of software evaluation, the aim can be summarized via the paradigm *specify once, evaluate any*. Translated to the field of process engine evaluation, this means that the steps to evaluate a particular feature of a process engine are only specified once. Those steps, however, are then used to evaluate any given engine, be it a new version of an existing engine, a different configuration, or a new engine altogether. In contrast to a company which may only work with a single process engine in production, a plethora of process engines have to be handled to reveal their differences. A step towards this goal is to use standard conformant process models within these evaluation steps. Since, however, each engine offers a different interface for the common management actions, and may even require engine-specific alterations of the standardized process models, there is a large gap between the status quo and the desired condition in which process engines can be evaluated easily.

Engine-Specific Steps

To evaluate a feature on an engine, one must first specify the corresponding process model along with the other necessary steps as part of the process design phase *once*. Evaluating a feature on a specific engine requires walking the three phases system configuration, process enactment, and diagnosis of the BPM lifecycle. Table 3.1 shows these steps grouped by their lifecycle phase in which they are taken. Namely, during system configuration, the process engine must be installed and started, and the process model converted to an engine-specific deployment package. Next, in the process enactment phase, the engine-specific deployment package has to be deployed to the started process engine so that

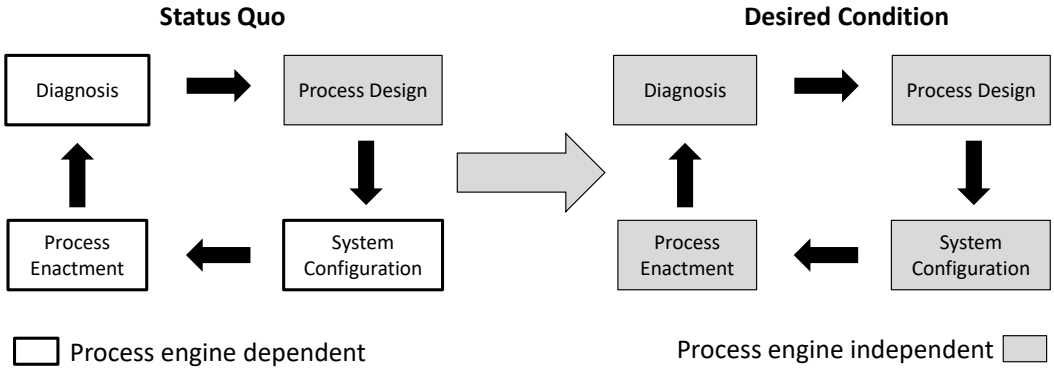


Figure 3.1.: The BPM lifecycle by [261] revealing the degree of engine independence at the moment and in an ideal situation [97, p. 2]

one can start and interact with instances of the previously deployed process model. Last, within the diagnosis phase, the states of the instances and the logs of the engine are required for evaluation purposes. Based on these engine-specific steps, it is proposed to derive a uniform API which provides these steps in an engine-independent manner with the engine-dependent behavior pushed to the internal engine-specific wrapper.

Table 3.1.: Engine-Specific Steps Grouped by BPM Lifecycle Phase

Phase in BPM Lifecycle	**Engine-Specific Interaction**
system configuration	install process engine
	start process engine
	create deployment package
process enactment	deploy deployment package
	start and interact with process instances
diagnosis	get state of instances

Chapter Structure

The remainder of the chapter is structured as follows. First, related work is presented in Section 3.2, followed by the design of the uniform API which is detailed in Section 3.3. Next, the prototype and its evaluation is detailed in Section 3.4 and Section 3.5, respectively. The chapter is concluded with a summary in Section 3.6.

3.2. Related Work

Uniform APIs

Uniform abstraction layers have already been used a lot in other domains such as for cloud computing providers [132, 165, 211], benchmarking multi-core performance [74], or accessing Web Services protocol-independently [52].

Uniform PaaS and IaaS APIs

In cloud computing, the interfaces of the products of the cloud vendors do differ a lot despite providing similar services [132, 133], especially for IaaS and PaaS offerings. Hence, the situation in cloud computing is similar to this work. But in contrast to the field of BPM, there are several uniform APIs available today for IaaS (e.g., the Open Cloud Computing Interface (OCCI) [185]) and for PaaS (e.g., OASIS's Cloud Application Management for Platforms (CAMP) [183] and nucleus [132, 211]).

Uniform Web Services API

Another uniform API is the Web Services Invocation Framework (WSIF) [52] which abstracts away from the protocols that can be used in conjunction with WSDL-based Web Services (e.g., SOAP [270]) through an abstract service and programming model. The developer can focus on the functionality by using the abstract programming model, and the protocol-specific parts are pushed down the stack where they can be optimized without touching the source code. This is similar to PEAL that abstracts from the engine-specifics in contrast to the protocol-specifics of the WSIF.

BPEL Management Framework

In the BPM domain, van Lessen et al. [265] propose a management framework for BPEL engines. First, they focus only on BPEL engines while this work takes BPMN engines into account. Second, they reveal the internal process models and their instances in an engine-independent manner through resources whereas this work focuses on building a uniform API for the engine itself without touching the internal models. And third, they provide ways to subscribe to changes in their uniform process and instance model through WS-Notifications [177–179], which is out of scope of this work. Their idea of handling process models and their instances in a resource-oriented fashion as in a REST interface is also available in the uniform API in this work. The process models and instances, however, are scoped by the engine in which they run whereas van Lessen et al. [265] have an implicit scope of the standalone engine that exposes their interface instead.

BPEL State Models

Both Kopp et al. [138] and Sonntag and Karastoyanova [242] present state models for BPEL process models and their activities. There are also state models available for the process models and their instances in PEAL. But Kopp et al. [138] and Sonntag and Karastoyanova [242] focus more on the state models of activities (i.e., internals of the process instances), leaving aside the state model of the process engine itself, which is covered by PEAL. When comparing the work by Kopp et al. [138] with that of Sonntag and Karastoyanova [242], it can be said that Kopp et al. [138] present a superset of the state models of Sonntag and Karastoyanova [242].

BPMN State Models

Delgado et al. [46] created a generic user portal API. This API covers the common task of interacting with users to get them to approve a task or enter input. For that, they provide a model of how to represent this user interaction. PEAL does not cover user interaction except to start a process instance. Hence, the API by Delgado et al. [46] is orthogonal to PEAL.

WAPI

The Workflow Management Coalition (WfMC) defined the Workflow APIs and Interchange Formats (WAPI) to standardize the communication within a WfMS and between applications and WfMSs. Those standards, however, are not applied by the BPEL and BPMN engines covered by this work. Hence, that API is not considered further in this work.

Uniform APIs in TOSCA

With the OASIS standard Topology and Orchestration Specification for Cloud Applications (TOSCA) [184] one can describe complex cloud application topologies as a typed graph based on node and relationship templates. The type of such node templates (i.e., the node type) also provides methods to manage their lifecycles [19, 21]. Typical methods of such a node type are *install*, *start*, *configure*, *stop*, and *isRunning*. A uniform API for BPEL and BPMN engines would be the perfect basis for building a node type of any BPEL or BPMN engine. The extension Lego4TOSCA [105] simplifies the creation of such node types with a uniform API as one can rely on the uniform API of other node types as well. For instance, a BPEL engine running on an Apache Tomcat could reuse the management API of the Apache Tomcat which in turn can reuse the

API provided by the underlying system, building upon an existing high-level API similar to Lego bricks.

The field of provisioning software on-demand is related as well. Dornemann et al. [50] provisioned BPEL process models on-demand alongside their process engine. A more general approach is presented by Vukojevic-Haupt et al. [268] which works for arbitrary workflow languages, and presents the roles and components required to ensure on-demand provisioning. Our work provides building blocks to implement on-demand provisioning. On-Demand Provisioning

As part of BenchFlow [61, 62], there is also a library[38] to abstract away the interaction with the systems under test (i.e., BPMN process engines). This library, however, only contains a deploy and a start process instance method, and it solely works for BPMN engines. BenchFlow

The UBML [97] is a preliminary version of PEAL that has been published earlier by the author of this work which did not include BPMN engines. PEAL supersedes the UBML by adding support for BPMN engines and a service to manage instances of process models. UBML

3.3. Design

In this section, the design of the uniform Process Engine Abstraction Layer (PEAL) is presented. First, the three core concepts, upon which the API of PEAL are designed, are described in Section 3.3.1, followed by the actual API design in Section 3.3.2. An example usage scenario of this API is outlined in Section 3.3.3 in the form of a service composition (i.e., a new service is composed on top of the designated API). This section concludes with the limitations of the chosen API and possible extensions in Section 3.3.4.

3.3.1. Concepts

PEAL abstracts away from the engine-specifics through a) making use of engine-independent *identifiers*, b) providing observers to reveal the *state* corresponding to each identifier, and c) handling engine-specific artifacts through different *packages*.

Each engine, process model, and instance of a process model is identified through its URI-like[39] Identifiers (IDs). The IDs reflect that an instance of a process model runs on a particular engine as the instance ID contains the process ID which in turn contains the engine ID, as shown on the left-hand side of Figure 3.2. The different services rely on these IDs to perform their actions appropriately and they also create and expose them. If appropriate, the engine-independent identifiers have to be mapped to the engine-specific Identifier

[38] `https://github.com/benchflow/sut-libraries`, visited 2017-3-31

[39] It only reuses the hierarchy separated by slashes. The other features of a URI are not used.

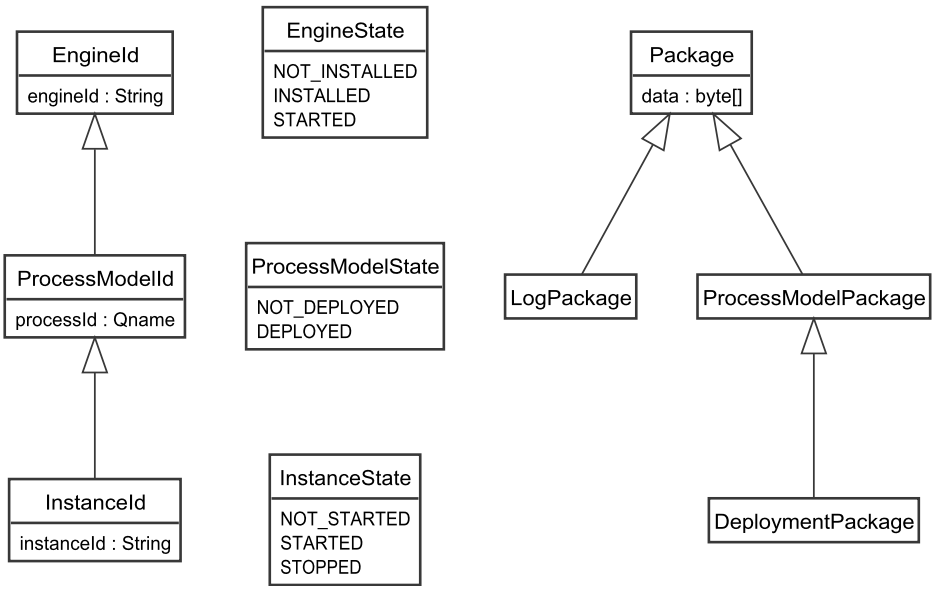


Figure 3.2.: UML Diagram outlining the three API Concepts of PEAL: *Identifiers*, *States* and *Packages* and their Relationships

ones internally. The engine ID and the instance ID are string-based, whereas the process model ID is based on a Qualified Name (QName) [274]. The latter consists of a name and a namespace, as it is typical for XML-based process languages such as BPEL and BPMN, and therefore a good fit in this case.

States

In addition to the typical methods that trigger an action, there is an observer to check the current state of the engine, process model or instance of the process model through its ID. As shown in the middle of Figure 3.2, an engine can either be *not installed*, *installed*, or *started*, and a process model is either *deployed* or *not deployed*. There are three possible states of a process instance: *not started*, *started*, and *terminated*. The terminated state itself can be more complex as there are many reasons why a process has terminated, but this is not relevant on this level.

Packages

When interacting with engines, files have to be transferred. These files can be engine-specific or contain engine-specific data. Such data has to be wrapped within a 'package'. For instance, there is the *process model package* containing the process model and all its related files, and the *deployment package* containing a deployable process model with all the files so that it can be deployed on a particular engine. For instance, a process model package of a BPEL process contains the WSDL and XSD files as well, while the deployment package of Apache ODE, for instance, contains the deployment descriptor in addition the engine-specific versions of the files in the process model package. Also, the *log package* contains engine-specific logs. This package can be used to retrieve the log data from the engine. All three packages and their relationship can be seen on the right-hand side of Figure 3.2. In this approach, a package is always an archive. Hence, the data of a package is stored in binary form (i.e., a byte array).

3.3.2. Uniform API

Based on the three scopes, namely, *engine*, *process model* and process *instance* (see Section 3.3.1 and Section 2.2.2), three services have been designed: the engine service, the process model service and the instance service. Their APIs are depicted in Figure 3.3. The methods are derived from the typical usages as listed in Table 3.1.

Services

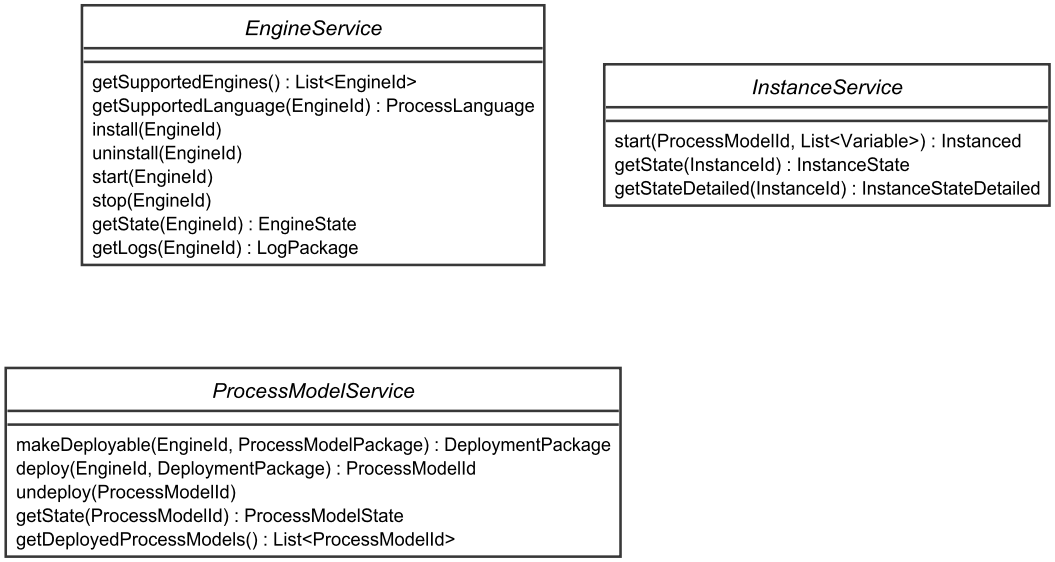


Figure 3.3.: The interface of the Engine, Process Model and Instance Service

Engine Service

The engine service (see Figure 3.3) comprises the provisioning of engines, managing their lifecycles, and obtaining the logs. It knows which engines it can provision and provides a list of IDs for these engines. Using one of these IDs, it is possible to install and uninstall the engine corresponding to a particular ID, or check whether it has already been installed. For the installation, a configuration can be passed to govern the installation itself, e.g., by providing the port where it should listen for requests. Moreover, any software required by the engine is either installed along the way on-demand or linked to according to a given configuration. The typical use case for linking to an existing and required software is that of the database. The target environment in which the engine is provisioned into is the one providing this API. Hence, only an environment in which the engines can be provisioned is allowed to provide this API. The lifecycle of the engines is controlled through starting and stopping an already installed engine and checking if an engine is already started. To assist in troubleshooting, all logs of an engine can be accessed in the form of a log package which acts as a log snapshot. This is necessary because a) an engine normally has more than a single log, b) the underlying container also has several logs, and c) the location of these logs differ from engine to engine, and from container to container.

Process Model Service

The process model service (see Figure 3.3) handles the deployment of process models. The deployment itself is a two-step process. In the first step, the process model package has to be transformed to a deployment package suitable for a specific engine. Subsequently, the deployment package can be deployed on this specific engine. What is more, the deployed processes can be queried, and, if need be, undeployed.

Instance Service

The instance service (see Figure 3.3) allows creating a new instance of a deployed process model. Starting a new process instance may require the passing of variables to initialize the state of the process instance. In addition to the high-level instance state, one can also get a more detailed instance state, revealing the cause of its termination. The instance service is solely required for BPMN as there is no standard how to start an instance of a process model, whereas for BPEL, one only needs to send a SOAP message to the appropriate WSDL endpoint [42].

3.3.3. Uniform API Composition

Composition

Based on the uniform API of PEAL, it is possible to build composite services on top through composition. An example is provided in Listing 3.1 which shows two methods, namely, *MAKE_AVAILABLE* and *MAKE_UNAVAILABLE*. The first one ensures that a *processModelPackage* is deployed onto a running engine identified by a specific *engineId* and the latter one reverses this by stopping and uninstalling the underlying engine if it is no longer required. Hence, both methods optimize the usage of resources (i.e., RAM and HDD usage), because resources are only allocated if necessary through on-demand install and start calls and, if possible, freed through stop and uninstall commands. These two methods represent typical use cases when working with engines. Thus, this service composition is a validation of the flexibility of the uniform API of PEAL.

Listing 3.1: Composite Service using the uniform API.

```
1  def MAKE_AVAILABLE engineId, processModelPackage:
2      if engineService.getState engineId is NOT_INSTALLED:
3          engineService.install engineId
4
5      if engineService.getState engineId is INSTALLED:
6          engineService.start engineId
7
8      let deploymentPackage = processModelService.makeDeployable engineId,
           processModelPackage
9      return processModelService.deploy engineId, deploymentPackage
10
11 def MAKE_UNAVAILABLE processModelId:
12     if processModelService.getState processModelId is DEPLOYED:
13         processModelService.undeploy processModelId
14
15     if processModelService.getDeployedProcessModels is empty:
16         engineService.stop processModelId
17         engineService.uninstall processModelId
```

3.3.4. Limitations

Limitations

The chosen API comes with limitations. First, this approach only works with deployment models containing exactly one process model and not more. This limitation has been chosen deliberately to avoid having to keep track of a deployment ID which complicates the ID tree further and to keep the API simple. For BPEL, it is straightforward as a single BPEL file can only contain a single BPEL process. In contrast, a single BPMN file can contain multiple BPMN process models. Second, the instance interaction service is simplistic since it supports only three operations. However, it is sufficient for the use case of this work, which is software evaluation and not instance management. Third, the management as well as monitoring of a process instance is not specified. Although there are many advanced instance management features such as suspend/resume, debug, recover, manipulate the state, and rewind [241], they are supported by a minority only, and not relevant for the typical usage steps according to Table 3.1. The same holds for translating the engine-specific history of a process instance to a generic audit trail.

3.4. Prototype

In this section, the prototypical implementation of the PEAL API is detailed. The prototype is written in Java 8, executable on Windows and Linux alike, and provides mappings of the uniform API to a plethora of BPEL and BPMN engines as shown in Table 3.2.

Technical Details

The API itself is exposed through Java 8 interfaces[40] natively and WSDLs 1.1 interfaces via JAX-WS 2.2.10 [34]. The prototype uses no internal state. The state is still within the underlying engines. Hence, it provides only a thin wrapper around the functionality of the engines.

3.4.1. Supported Engines

Engines

The prototype covers three different BPMN engines in five to six versions each, and seven BPEL engines in different versions and configurations. The most recent version of each of the eleven engines was picked on August, 28th 2016. These engines were not picked at random, but selected by collecting lists of all engines stating that they implement the process language, and then reducing that list down to the presented engines by evaluating the following criteria for each engine on the list: 1) the engine is open source and freely available, 2) the engine understands process models in the standard conformant serialization format, and 3) process models can be deployed automatically. Moreover, these engines are actively maintained and rated as mature projects by their respective vendors. Consequently, these engines are representative to show that the

[40] `https://github.com/uniba-dsg/betsy/tree/master/peal`, visited 2017-3-31

uniform API of PEAL can be mapped to any process engine implementing BPEL or BPMN. Each engine is described shortly in the following. For an overview of their licenses, release dates, or the programming language they are developed in, see Appendix A.

Table 3.2.: Supported Engines with their Configurations by the PEAL Prototype

process language	engine name	engine version	configuration
BPMN	Activiti	5.15.1, 5.16.3, 5.17.0, 5.18.0, 5.19.0.2, 5.20.0, 5.21.0, 5.22.0	
	jBPM	6.0.1, 6.1.0, 6.2.0, 6.3.0, 6.4.0, 6.5.0	
	camunda BPM	7.0.0, 7.1.0, 7.2.0, 7.3.0, 7.4.0, 7.5.0	
BPEL	ActiveBPEL	5.0.2	
	bpel-g	5.3	in-memory
	Petals ESB	4.0, 4.1	
	Apache ODE	1.3.5, 1.3.6	in-memory
	OpenESB	2.2, 2.3, 2.3.1, 3.0.1, 3.0.5	
	Orchestra	4.9	
	WSO2 BPS	2.1.2, 3.0, 3.1, 3.2, 3.5.1	

Activiti

"Activiti is an open source BPM platform. In 2010, developers already working on jBPM decided to build a new BPM engine from scratch exclusively designed for BPMN execution and this engine is the result. Activiti is supported by industry, however most core developers are associated with Alfresco who provides an enterprise edition of Activiti." [81, p. 25]

camunda BPM

"[The process engine] camunda BPM is a fork of Activiti which is now developed and distributed by the German BPM software vendor camunda. Next to the open source version of camunda BPM, also an enterprise edition is available." [81, p. 25]

jBPM

"Originally, jBPM was not developed as a distinguished BPMN engine but as a more general BPM platform. Based on a Process Virtual Machine (PVM), it supports several process languages (e.g., JPDL and BPEL). Since version 4.3, jBPM also supports the execution of processes in native BPMN. " [81, p. 25]

Active-BPEL

The open source BPEL engine ActiveBPEL is the open source variant of the same-named product ActiveBPEL by Active Endpoints. It has been released in version 5.0.2 without any tests in 2008. Since then, Active Endpoints did not contribute any code to the open source variant, but only to their proprietary product.

bpel-g

"The bpel-g engine is a derivate of the former ActiveBPEL by Active Endpoints. Whilst ActiveBPEL is no longer available, bpel-g is still under development and maintained as a Google Code project. The engine comprises the functionality provided by ActiveBPEL, but is extended to support and integrate with software libraries, such as Spring." [94, p. 5]

"Petals ESB is an open source Enterprise Service Bus (ESB) that includes a BPEL service engine and a SOAP binding component. It is developed by the OW2 consortium, just as Orchestra, and is available at `http://petals.ow2.org/`. Instead of reusing Orchestra as a BPEL engine, Petals ESB provides a separate engine, namely EasyBPEL. [...] Just like the other engines, EasyBPEL is written in Java." [94, p. 6] Petals ESB

"As of today, Apache ODE is the most well-known and most widely used Open source BPEL engine available. It is maintained by the Apache Foundation and supported among others by Intalio and JBoss. The engine is implemented in Java and relies on Jacob, a concurrency framework based on the actor model [109]." [94, p. 5] Apache ODE

"The OpenESB is an open Enterprise Service Bus that includes a BPEL engine. It is written in Java and preceding its acquisition by Oracle, it was maintained by Sun. Today, its development is driven by LogiCoy and Pymma Consulting. OpenESB is commonly collocated with the Glassfish application server to form a full enterprise integration solution. [...]." [94, p. 5–6] OpenESB

"Orchestra is [...] written in Java and developed by the OW2 consortium and Bull. [...] Orchestra executes BPEL on a generic process virtual machine. [..] Orchestra does not require a separate deployment package for deploying to an engine. Instead, it is sufficient to provide the BPEL and WSDL files directly. Although not being required, it is still possible to use a packaged format [194, pp. 21/22]." [94, p. 6] Orchestra

The WSO2 Business Process Server (WSO2 BPS) is an open source BPEL and BPMN engine. Internally, it uses Apache ODE for executing BPEL processes and Activiti for executing BPMN processes. Although it does not implement a process engine by itself, it is included for PEAL because it builds so much software around Apache ODE that its behavior is quite different in comparison to the standalone Apache ODE. In contrast, the API of Activiti is exposed directly[41] instead of hiding the used BPMN engine behind another set of software layers as it has been done for Apache ODE. WSO2 BPS

3.4.2. Engine-Specific Mappings

Each engine requires a custom mapping from its engine-specific API to the uniform engine-independent API of PEAL. The differences are present in the actual implementation, but also classified in Table 3.3 for the BPMN engines, and in Table 3.4 for the BPEL engines. The engine-specifics are grouped into installation, deployment, and process language specifics. Engine-Specifics

Every engine is implemented in Java and runs within some sort of managed environment (i.e., container as in runtime environment of a specific Java application instead of Docker container for arbitrary applications). Because the engines have a similar setup, it is possible to compare their setup to some Installation

[41] `https://docs.wso2.com/display/BPS350/BPMN+REST+API`, visited 2017-3-31

degree and show which Java version and which container each engine needs. If additional software apart from the container and the JRE is required, it is listed as well. When there is a deployable WAR file, which just needs to be deployed on a suitable container without any installation routines, it is indicated, too.

Deployment

The deployment itself consists of the deployment package and the deployment method with which a deployment package can be deployed to an engine. The deployment package has a specific file extension and contains additional files on top of the process model that adhere to a specific format. The deployment methods are either remote REST calls, local CLI calls, or local File System (FS) interaction. The local FS interaction works by copying a file to a specific place and waiting for a log to indicate the outcome of the deployment in the form of a specific log entry. The term for this type of action is Failable Timed Action (P19). Such deployment methods can also differ in between different versions of the same engine, e.g., the REST API could change its result from one version to a newer one. The file extension of the deployment package can either be the format of the process model itself (e.g., BPMN), or a package containing multiple files (e.g., a JAR, WAR, BPR, or ZIP file). The format within these files can either be engine-specific and define a standard for other engines, or conform to a more open standard such as Java Business Integration (JBI) or BPR with its Business Process Archive Descriptor (BPRD). The number of additional files is based on the format of the deployment package.

Process Language Specifics

Since BPMN does not rely on as many other standards as BPEL, there are more engine-specifics for BPMN engines that have to be taken into account. As BPMN supports the *scriptTask* which can execute a script defined in a specific programming language, one needs to know which engine supports which programming language for the scripts. Moreover, engines do differ in how the variables in a process need to be set up [81]. Some require that variables have to be set explicitly, and others require that they should only be used implicitly instead.

Other Engine-Specifics

The engine-specifics mentioned above are the most important and relevant ones for implementing the uniform API. Nevertheless, other engine-specifics such as the format of log messages are relevant as well. Such engine-specifics are described in the text and are not listed in any of the tables below.

jBPM

As shown in Table 3.3, the three BPMN engines vary greatly. The BPMN engine jBPM runs on the JBoss application server 7 and relies on an Apache Ant build script for its installation and startup procedures. It does not yet support Java 8. Hence, Java 7 must be supplied. The deployment is complex, as one needs to 1) build the JAR file with its jBPM specific deployment descriptors, 2) install the built JAR into the local maven repository via a CLI command, and 3) issue a REST call which deploys the JAR from the local maven repository. In contrast to Activiti and camunda BPM, jBPM requires that all variables in the process must be modeled explicitly with jBPM specific data types, and it

Table 3.3.: Engine-Specifics of the BPMN Engines, based on [81]

	engine versions	jBPM all	Activiti all	camunda BPM 7.0.0	cam. BPM >= 7.1.0	cam. BPM >= 7.4.0
install	container	Jboss AS 7	Tomcat 7	Tomcat 7	Tomcat 7	Tomcat 8
	requirements	ant, maven		maven	maven	maven
	Java Version	7	8	7	8	8
	deployable war		x	x	x	x
deploy	method	CLI + REST	REST	FS	FS	FS
	file extension	JAR	BPMN	WAR	WAR	WAR
	file format	JBPM	-	CAMUNDA	CAM.	CAM.
	additional files	3	0	3	3	3
	scripting engine	Java	Groovy	Groovy	Groovy	Groovy
	process var.	explicit	implicit	implicit	implicit	implicit

can only execute script tasks written in Java, not in Groovy. What is especially problematic is that the REST API is not stable and changes between versions.

activiti

Of the three BPMN engines, Activiti has the easiest setup as it only requires the presence of an Apache Tomcat 7 container, onto which the WAR file has to be deployed. And Activiti works on both, Java 7 and Java 8. The deployment is the simplest as well, with passing a BPMN file to the engine through a REST call – no deployment descriptors or any other additional file necessary. Activiti supports the scripting engine Groovy, and allows that the variables in the process can be defined implicitly through the Groovy scripts themselves.

camunda BPM

The complexity of the maven-based installation of camunda BPM is in between that of jBPM and activiti. It requires an Apache Tomcat 7. Camunda BPM 7.0.0 requires Java 7, but Java 8 is supported for 7.1.0 and later. Starting with camunda BPM 7.4.0, Tomcat 8 is required as well. The deployment is the same for all versions of camunda BPM, as it only requires that a WAR file has to be built using a camunda-specific format, and then deployed via the FS interaction for all the versions. The step to undeploy the process has to be done via the FS-based method for camunda BPM 7.0.0 and 7.1.0. But starting with camunda BPM 7.2.0 the REST API is capable of performing this task as well. Regarding the process language specifics, it behaves similarly to activiti.

Apache ODE and bpel-g

Analogous to the BPMN engines, the BPEL engines do differ a lot as well as shown in Table 3.4. The exception from this rule are Apache ODE and bpel-g, which are similar as both require an Apache Tomcat 7 and Java 8, and both can be deployed through a WAR file. Also, they use similar deployments: bpel-g not only reuses the deployment format of Apache ODE but also offers a FS-based deployment method similar to Apache ODE.

OpenESB

OpenESB has one of the most complex installation procedures and deployment routines. The installation of OpenESB up to version 2.3.1 requires Java 7 and a full blown application server called Glassfish v2. The installation has

Table 3.4.: Engine-Specifics of the BPEL Engines, based on [94, 150]

	engine versions	Apache ODE all	bpel-g all	OpenESB <= 2.3.1	OpenESB >= 3.0.1
install	container requirements	Tomcat 7	Tomcat 7	Glassfish v2	OpenESB SE 3
	Java Version	8	8	7	8
	deployable war	x	x		
deploy	method	FS	FS	CLI	CLI
	file extension	ZIP	ZIP	JAR	JAR
	file format	ODE	ODE	JBI	JBI
	additional files	1	1	3	3

	engine versions	Orchestra all	Petals ESB all	WSO2 BPS all	ActiveBPEL all
install	container requirements	Tomcat 7 ant	Petals ESB 4	WSO2 Carbon	Tomcat 5
	Java Version	8	7	8	7
	deployable war	x			
deploy	method	CLI	FS	FS	FS
	file extension	ZIP	JAR	ZIP	BPR
	file format	-	JBI	ODE	BPR
	additional files	0	3	1	1

to be triggered through its CLI-based installation tool. Starting with version 3.0.1, OpenESB works with Java 8 and uses a much more lightweight container called OpenESB SE 3. The deployment method, however, has stayed the same: one needs to build a JAR based on the JBI format and deploy it using CLI commands, but these commands differ between OpenESB 2.x and 3.x.

Orchestra

To install Orchestra, an Apache Ant build script has to be executed which installs Orchestra on a Tomcat 7 using Java 8. The deployment is also based on an Apache Ant build script and it requires a simple ZIP file containing the process model and its necessary files. It requires the simplest deployment package without any deployment descriptor.

Petals ESB

The installation of Petals ESB is straightforward and requires Java 7. The BPEL engine is installed as a component of the Petals ESB 4. Petals ESB is as complicated as OpenESB in its deployment, as it also relies on the JBI standard, and it requires the more complicated FS deployment method.

WSO2 BPS

The WSO2 BPS runs on Java 8, and requires the ESB called WSO2 Carbon as its infrastructure. The deployment is based on the deployment of Apache ODE because WSO2 BPS uses Apache ODE internally.

Active-BPEL

The ActiveBPEL engine still relies on Tomcat 5 and is only executable on Java 7. The deployment is based on the BPR which relies upon the BPRD.

BPEL Engines

Also, each BPEL engine needs to set the endpoint address in the WSDL endpoint definition to reflect the location of the service. Moreover, both bpel-g

and ActiveBPEL need to configure the underlying servlet container to be able to run on them, and bpel-g has to remove all unimplemented (i.e., unbound) operations of the WSDL services the BPEL process model is implementing.

Proprietary Engines

Proprietary engines can be mapped to the uniform API of PEAL as well. Three proprietary BPEL engines have been interacted with in a uniform manner in an early version of PEAL [96]. Because of licensing restrictions, however, the mappings cannot be published and are not further detailed in this work.

3.4.3. Implementation

The mapping of the uniform API to the engines themselves is implemented manually in Java. In case the method of the uniform API can be mapped to one or more calls to an engine-specific REST API, the mapping is easy to achieve. Often, however, the interaction requires FS interaction, parsing of logs or calling custom CLI tools, or executing shell/batch scripts. As some actions may take some time, it is difficult to know when the action has been completed and whether it succeeded or failed. To solve this, the prototype waits for a specific amount of time after each action wherein it checks both, the condition for a success and a failure. If neither of the conditions is reached within that time, a timeout occurs and the action is marked as failed (see Failable Timed Action (P19)). Most conditions rely on low-level log messages. Hence, the log level had to be adjusted to provide Detailed Logs (P17). Moreover, as timeouts can either be too small or too large, they also have to be calibrated (see Timeout Calibration (P20)). What is more, XSLT scripts are used to enrich engine-independent files with engine-dependent information and to generate deployment descriptors.

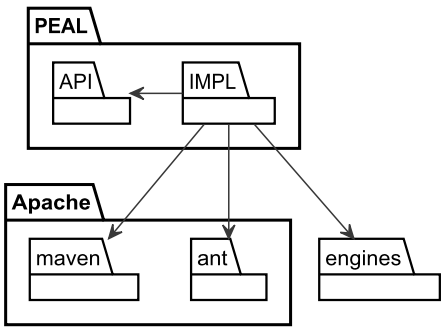


Figure 3.4.: The Architecture of PEAL

Architecture

All software necessary to install an engine is loaded and installed from the Internet on-demand. Hence, the only dependency of this prototype is Java 8, making the use of this prototype straightforward in other software. The architecture of the prototype is depicted in Figure 3.4. It reiterates that there is an API and an implementation of it based on the engines using Apache Ant and Apache Maven.

3.4.4. Limitations

The prototype contains various shortcomings and limitations. First, the creation of deployable packages only works for specific BPEL and BPMN process models, and not for every process model of those two languages so far. With more sophisticated deployment descriptor generators and deployment package builders, a much larger set of process models could be handled. Second, as the prototype cannot pass configuration options to the engines, each process engine uses its default ports which may conflict with ports of other software and other installed process engines. Hence, only one process engine can run at the same time. This can be mitigated by incorporating isolation through Linux Containers or VMs. Third, not every engine supports the undeploy step. This, however, is not critical for this work as the process engine is stopped and uninstalled for each test anyway, effectively undeploying the process model on the go. Fourth, the instance service is only implemented for BPMN engines. It may be useful for BPEL engines as well. Despite these shortcomings, it has been shown that the goal of creating a uniform API for these management tasks BPM lifecycle is feasible, but an industry ready implementation of this API requires more resources.

3.5. Evaluation

The prototype from Section 3.4 has been evaluated by conducting the following evaluation procedure shown in pseudo code in Listing 3.2. This procedure reflects the typical scenario based on the BPM lifecycle and incorporates all the engine-specific steps taken during benchmarking as shown in Table 3.1. It is based on the service composition methods *MAKE_AVAILABLE* and *MAKE_UNAVAILABLE* as shown in Listing 3.1. After each line of the evaluation procedure, the appropriate assertions check the state with the relevant IDs to verify that the result is correct. By performing the evaluation procedure on both Linux and Windows it is ensured that the prototype is cross-platform. For the BPEL and BPMN use case, the most simplistic process there is used: the Aptitude Test (P8). Hence, if this test works, the engine fulfills the minimal criteria for implementing this uniform API.

Listing 3.2: Use Case Based Evaluation Procedure

```
for each engineId in engineService.getSupportedEngines:
    let supportedLanguage = engineService.getSupportedLanguage engineId
    if supportedLanguage is BPEL:
        let processPackage = bpelPackage
    if supportedLanguage is BPMN:
        let processPackage = bpmnPackage
    let processModelId = makeAvailable engineId, processModelPackage
    if engineService.getSupportedLanguage engineId is BPEL:
        // send soapMessage
    if engineService.getSupportedLanguage engineId is BPMN:
        instanceService.start processModelId, bpmnVariables
    makeUnavailable processModelId
```

For BPEL, a simple echo process as shown in Listing 3.3 is used which receives a SOAP message containing an integer, copies the integer to the response, and returns the response as another SOAP message. It is tested by sending a SOAP message containing the number 1 and asserting that the response also contains the number 1. BPEL Process

Listing 3.3: BPEL Process with SOAP Message Pair Used for Evaluating the Uniform API Implementation of the BPEL Engines

```
<process ...>
    <import ... />
    <partnerLinks> ... </partnerLinks>
    <variables> ... </variables>
    <sequence>
        <receive createInstance="yes" .../>
        <assign ... />
        <reply ... />
    </sequence>
</process>

<!-- request message -->
<Envelope>
    <Header/>
    <Body>
        <testElementSyncRequest>1</testElementSyncRequest>
    </Body>
</Envelope>

<!-- expected response message -->
<Envelope>
    <Header/>
    <Body>
        <testElementSyncResponse>1</testElementSyncResponse>
    </Body>
</Envelope>
```

For BPMN, on the other hand, a process containing only a single script task which is linked from the start event to the end event as a simple sequence through sequence flows is used. The script task writes a specific execution trace as shown in Listing 3.4. In the experiment, it is asserted that the expected execution trace is produced by the previously described BPMN process (i.e., that the execution trace contains the element “task1”). BPMN Process

Listing 3.4: BPMN Process with Expected Execution Trace Used for Evaluating the Uniform API Implementation of the BPEL Engines

```
<definitions>
    <process isExecutable="true" ...>
        <startEvent ... />
        <sequenceFlow ... />
        <scriptTask ... /><!-- writes execution trace 'task1' -->
        <sequenceFlow ... />
        <endEvent ... />
    </process>
</definitions>

<!-- expected execution trace -->
task1
```

The evaluation itself is done on two different systems. First, it has been conducted on a Dell Latitude E6340 laptop with an Intel i7-3520M@2.90GHz with 8 GB DDR3 RAM running Windows 10 Pro 64-bit. Second, the evaluation is conducted on Travis CI, a cloud-based continuous integration provider which executes builds in Docker containers. Those containers run an Setup

Ubuntu 12.04.5 LTS 64-bit operating system based on the 3.13.0-29-generic kernel. The procedure has been implemented as JUnit tests[42] for both BPMN and BPEL engines. On the local desktop machine, the tests are executed via a CLI command. For Travis CI a special configuration file[43] is created which triggers the same command as on the desktop machine but also defines the environment the prototype is running in.

Results

The results of the evaluation conducted on Travis CI are available online[44]. It shows that the prototype and its API are suitable to perform the engine-specific tasks behind an engine-independent abstraction layer. The assertion after each line returned the expected results, as the tests are all green. The results of the evaluation on the Windows laptop are not available online, but they produced the same results as the ones on Travis CI. Hence, the evaluation procedure works on both Windows and Linux.

Threats to Completeness

Three different BPMN process engines and seven different BPEL process engines are covered by this prototype and its evaluation. For both process languages, there are more process engines available than only those that have been integrated into the prototype. However, there are reasons why it was not possible to integrate more. Regarding BPMN, the reason is that the other engines do not fulfill the Aptitude Test (P8) [81, 84, 85]. Only three out of the evaluated 47 BPMN engines fulfill the criteria to be integrated into this API. The reasons are manifold, including products that have only modeling and no execution support, no support for the native serialization of BPMN process models, or the licenses restricts one from benchmarking and from publishing any benchmarking results. For the BPEL engines, the case is somewhat different. In contrast to BPMN, fewer implementations exist for the BPEL specification. Moreover, many BPEL process engines are commercial ones which come with a licensing agreement that forbids benchmarking and the publishing of benchmarking results as well. The ones that only support the older BPEL 1.1 or that seems to be unsupported are neglected, and the remaining engines are the seven engines which are used in this prototype.

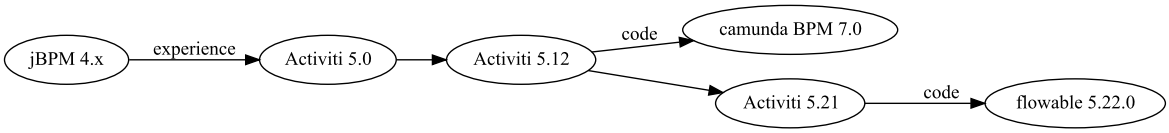


Figure 3.5.: The Fork-based Evolution of the BPMN Engines

Threats to Validity

The BPMN engines camunda BPM and Activiti are not entirely different process engines. The first one is a fork from the latter. As shown in Figure 3.5, the development began with jBPM. Activiti 5.0 has been developed

[42] https://github.com/uniba-dsg/betsy/blob/master/src/test/groovy/peal/impl/, visited 2017-3-31

[43] https://github.com/uniba-dsg/betsy/blob/master/.travis.yml, visited 2017-3-31

[44] https://travis-ci.org/uniba-dsg/betsy/builds/163700421, visited 2017-3-31

by the lead developers who already had developed jBPM 4. Hence, the experience in developing jBPM 4 has been used for building Activiti 5. The engine camunda BPM 7.0 is an actual fork[45] of Activiti, starting with Activiti 5.12.0 in March 2013. However, since 2013, both camunda BPM and Activiti have diverged in their capabilities [81, 84, 85]. This shows that despite having shared code at some point in time, they have evolved separately, and have resulted in different engines. Activiti 5.21.0 has been forked[46] again in October 2016 to flowable 5.22.0. As flowable 5.22.0 is only a rebranded variant of Activiti 5.21.0, it is not included in this study.

Testability and Installability

The effort of mapping the uniform API of PEAL to an engine can be roughly assessed based on the prototype and its evaluation. This allows making assumptions about the testability and installability of those engines, as was done by Lenhard et al. [150]. But this, however, is out of scope of this work.

3.6. Summary

Summary

This chapter outlines that there is the need to interact with process engines through a uniform interface, proposed such a uniform API, and evaluated it and its prototype through use cases. The API consists of services for managing the lifecycle of process engines, process models deployed on the engines, and instances of those process models that are being executed by engines. The prototype is able to implement the API for seven BPEL engines and three BPMN engines. The evaluation showed that the API with its prototype is sufficient to handle the typical scenarios according to the BPM lifecycle. Hence, hypothesis H4.1 *("A uniform interface is a suitable solution to interact with widely different process engines in a similar way.")* is supported.

Future Work

Future work comprises 1) creating a TOSCA process engine node type, 2) a unified monitoring and audit trail service, and in the long run 3) moving Business Process as a Service (BPaaS) [162] forward. A TOSCA process engine node type is desirable because it would allow to use the PEAL API within the TOSCA ecosystem. And this, in turn, would allow simplify working with process engines further. A unified monitoring and audit trail service is planned for future work as well. This is in line with the effort to provide standards and uniform APIs for the different phases within the BPM lifecycle. Moreover, this would help to provide data for the uprising field of process mining [49] as well. Last, bringing such an API into the cloud as a service would help in bringing BPaaS [162] forward. The client needs to define his business processes in the form of process models and then can use the infrastructure of a cloud provider to bring his process to life.

[45] `http://www.jorambarrez.be/blog/2013/03/18/a-sad-day-for-open-source-camunda-decides-to-fork-activiti/`, visited 2017-3-31

[46] `http://www.flowable.org/blog/2016/10/12/flowable-and-activiti.html`, visited 2017-3-31

> The not surprising and very obvious point is that every software project becomes a DSL.
>
> *Bob Bockholt*

4. Process Engine Benchmark Language

Parts of this chapter have been taken from [81–84, 94–96, 98, 99].

In this chapter, hypothesis H4.2 *("A domain-specific testing language is a suitable form to make quality criteria measurable.")* is supported.

This chapter presents the Process Engine Benchmark Language (PEBL) which is used to express benchmarks and their results. Such benchmarks can be conducted with the Process Engine Benchmark Framework (PEBWORK) described in Chapter 5 to produce results. Both benchmarks and results can be visualized in the Process Engine Benchmarking Interactive Dashboard (PEBDASH) presented in Chapter 6.

4.1. Motivation

Representation Requirements

According to Huppler [111] and Sim et al. [229], a good benchmark has to fulfill different criteria as described in Section 2.3.2. The subset of those criteria which drive a good benchmark representation are: *reproducibility*, *clarity*, and *portability*. Within a benchmark, the steps how it should be performed must be stated. Otherwise, the results of the benchmark cannot be reproduced. Moreover, the representation must be concise, self-contained, and to the point to be clear about its intent. Last, the benchmark should be abstract enough so it can be conducted for different implementations without any changes. Hence, it must be independent of the system under test.

Solution: DSL

In this work, a testing Domain-Specific Language (DSL) is proposed which aims to fulfill the three criteria mentioned above. A testing DSL can use domain-specific terminology to provide a clear and concise description of the steps necessary to execute the benchmark. The suggested terminology is a combination of existing terminology from the testing (see Section 2.3.3.1) and the BPM domain with its process languages, models, and engines (see Section 2.2.2). The DSL provides a serialization of a benchmark. Based on such a serialized benchmark, quality characteristics of the process engines should be revealable. In other words, hypothesis H4.2 *("A domain-specific testing language is a suitable form to make quality criteria measurable.")* is put under scrutiny.

The main requirement of a language for the specification of benchmarks and their results is *expressiveness* (see Section 2.2.3). This means that the elements in this language can be directly used to express a benchmark with its necessary steps. The more language elements are required to express single concepts in the domain of benchmarking process engines, the less expressive it is. According to this characteristic, the proposed DSL will be evaluated. Expressiveness

The remainder of the chapter is structured as follows. First, related work is presented in Section 4.2, followed by the DSL itself in Section 4.3. The prototype of the DSL can read, write, validate, and work with serializations of the DSL and is described in Section 4.4. In Section 4.5, the language is evaluated whether it can express a variety of different benchmarks. Chapter Structure

4.2. Related Work

There are approaches and tools available for testing BPEL-based and BPMN-based process models that come with their own testing or benchmarking DSL. Those approaches are either centered around unit or performance testing. In the following, those approaches are discussed, starting with the ones doing unit testing followed by the ones doing performance testing. Testing DSLs

Tools like soapUI[47] or BPELUnit [161, 168] have been developed to automate unit testing of SOA-based systems and BPEL engines, respectively. They both have a DSL in which tests can be specified. To describe their results, they build upon XML reports of JUnit. These languages share some concepts with PEBL, such as the availability of tests, test cases, test partners, test steps, and test assertions. PEBL, however, is more general as it allows expressing benchmarks with aggregated metrics for different process languages and multiple capabilities. A comprehensive overview of academic approaches to WS testing is given by Bozkurt et al. [26], but is out of scope as it does not directly address testing process engines. Unit Testing DSLs

Similar to the unit testing tools and approaches, there are also performance testing tools and approaches that also come with a DSL, such as GENESIS2 [123, 124], SOABench [16, 17], BenchFlow [61], Benchmark DSS [236], and DSLBench [29]. The first two are rather generic and set in the middleware or SOA performance testing domain. As they are performance testing tools, much of the DSL is about defining the environment in which the performance is later evaluated as well as the modeling of the user-based workload. BenchFlow [61] also uses a DSL[48] to describe benchmarks and tests. The benchmark in Section 4.5.3.4 shows that PEBL can cover their DSL through the use of the extension element. Skouradaki et al. [236] present a conceptual model as part of a Decision Support System (DSS), which can be seen as a benchmark DSL. Performance Testing DSLs

[47] `https://www.soapui.org/`, visited 2017-3-31
[48] `https://github.com/benchflow/benchflow`, visited 2017-3-31

This DSL represents performance benchmarks and their results. It is similar to the DSL from BenchFlow [61], as it is from the same authors as BenchFlow. The conceptual model of the DSS is the minimum set of components a benchmark for process engines should contain. A subset of this conceptual model is presented as part of BenchFlow [62, Figure 2]. PEBL is, in contrast, a more general DSL as it covers benchmarks for multiple quality characteristics, and does not contain language elements for describing performance benchmark specific constructs directly. Furthermore, as in performance benchmarking it is necessary to evaluate a complete system under test, the DSL of BenchFlow takes the topology of the system under test into account, including the database, underlying hardware, software, and the network in between the different nodes in the system topology. These elements are not part of PEBL. What is more, Bui et al. [29] present DSLBench, a performance benchmarking DSL which can be used to generate the code of a benchmark application specific to a particular system under test. The actual benchmark is executed through the generated benchmark application.

Test Terminology

The test terminology is defined in glossaries [87, 203] and standards [117] and used in a plethora of testing tools (e.g., soapUI and JUnit) with slight variations of its meaning. This work builds upon the notion of the IEEE Standard Glossary [203] that is similar to the ISO/IEC 25051 standard ISO/IEC [117]. The terminology is described in Section 2.3.3.1.

4.3. Domain-Specific Language

Entities

The Process Engine Benchmark Language (PEBL) is a Domain-Specific Language (DSL) that can be used to express both, process engine *benchmarks* and their corresponding *results*. Consequently, the language itself is structured into these two parts. Benchmarks are defined in the form of *tests* for *features* of *capabilities* as shown in Figure 4.1. A feature refers to a measurable part of one capability of a process engine. It is grouped via a tree-like structure to provide an aggregation hierarchy within metrics can be defined. A test consists of steps to reproducibly evaluate a specific feature on any engine. The results build upon both, tests and features, as a test result describes the results of having evaluated a feature by executing a test on a particular engine. The *aggregated results* reference the features and provide measurements of the previously defined metrics within the aggregation hierarchy. The arrows in Figure 4.1 describe the high-level dependencies between these elements. The elements are identifiable through their IDs. This allows describing each element separately in detail, as the only connection in between them is the ID, and persisting each element separately, allowing them to be easily exchanged and reused.

V-model

PEBL is based on a V-model containing two axes, namely, the benchmark completeness (left-to-right) and level of abstraction (top-to-bottom), as shown in Figure 4.1. To complete a benchmark, the path of the letter V is followed. We

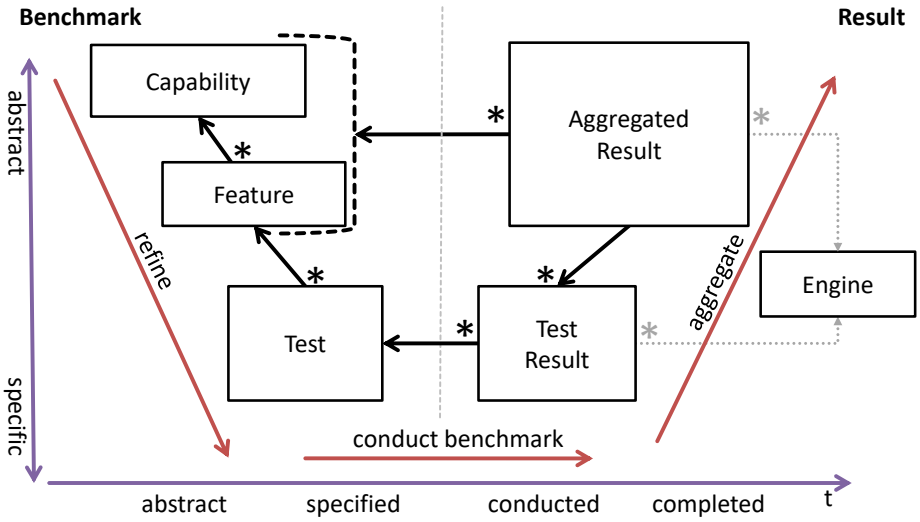


Figure 4.1.: The V-Model of PEBL

start by defining *abstract* features which are refined to more specific ones down to actual tests. As a result, we have *specified* the benchmark. This benchmark can then be *conducted* (i.e., executed) on a specific engine, producing test results. Afterwards, these low-level test results can be aggregated again to aggregated results and even abstract aggregated results representing special capabilities of the engines, creating a *completed* benchmark. The aggregation itself is done along the line for a particular engine as we are interested in the aggregated results of an engine and its characteristics.

V-model from Software Engineering

The term V-model is originally coined by Rook [214] in the field of software engineering. It is based on the concept that each phase in the waterfall model requires a corresponding testing phase. This creates a V-like shape because the steps from the waterfall model start abstract from specifying the requirements and becoming more specific down to the implementation phase which produces something that can be tested via unit tests up to acceptance tests to ensure that the specified requirements are fulfilled. Mathur and Malik [167] have extended the classical V-model to an advanced V-model which also incorporates software maintenance. The V-model in this work is based on the idea that each specification with its metrics can have results that provide measurements for the previously defined metrics, and both, the specifications and the results can be expressed at different levels of abstraction.

Roles

The presented V-model requires three different roles: business analysts, developers, and a benchmark framework. On the left side of the V-model (i.e., at the beginning), a business analyst specifies the capabilities and required metrics necessary to determine the quality of an engine. Next, a developer refines the features to actual tests and describes how to map the test results back to the required metrics of the features. Last, a benchmarking framework can then compute the results based on the specified benchmark, including the computation of aggregated metrics for the features. Hence, the benchmark

specification has to be done by humans whereas the results computation and their aggregation can be automated.

Syntax and Semantics

According to Fowler [69] there are internal and external DSLs. PEBL is an external DSL because it has its own syntax and does not depend on a host language. The only other language PEBL uses is Groovy to express user-defined executable scripts. These scripts, however, are expressed as strings in PEBL, effectively ignoring the semantics of Groovy to keep the complexity of PEBL manageable. PEBL has two serialization formats: XML and JSON. For both, detailed and specific grammars are specified through the language-specific schema definition languages: XSD and JSON schema[49]. The schema definitions of PEBL can be used to determine whether a given XML or JSON structure is a valid PEBL serialization.

Extension

The structure of PEBL allows expressing benchmarks and their results through its predefined domain-specific elements and attributes. If these domain-specific elements do not suffice to express a specific benchmark and its results, there are two possibilities to cope with this situation. The preferred option is to use the key-value stores available for each major language element that allows storing any additional information. An example of this is available in Section 4.5.3.4. There, it is shown that PEBL can express the performance benchmarks only with the help of these key-value stores. Moreover, some elements such as the test step *execute script* acts as a generic test step which can be used as an extension for a custom test step as well. The other option is to extend the language itself, creating a more expressive version of PEBL. The downside, however, is that any tool which needs access to the extension elements requires an extension to its parser and validation. Hence, the second option should be used with care.

In the following, each element of PEBL is described in more detail starting with the benchmarks in Section 4.3.1 followed by the benchmark results in Section 4.3.2.

4.3.1. Benchmarks

The benchmark specification of PEBL is subdivided into the feature tree and the tests. A test is used to determine whether a feature is supported. They are detailed in the inverse order of their dependency.

Features

The UML class diagram in Figure 4.2 shows the classes available to describe a feature. Features, feature sets, groups, languages, and capabilities each have an ID, a name, and a description. Moreover, they can be extended through their key-value store. Together, they form a tree-like structure with a capability as the root, the language, group, and feature sets as nodes, and the features as leafs: the *feature tree*. The link to the quality model of the ISO/IEC 25010 standard [113] has to be set at the capability level through the characteristics

[49] `https://github.com/uniba-dsg/betsy/tree/master/pebl/src/main/resources/pebl`, visited 2017-3-31

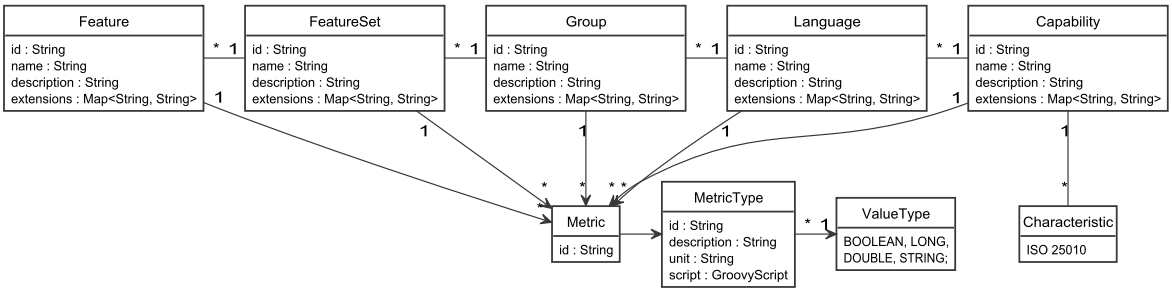


Figure 4.2.: The Feature Tree of the Benchmark Specification of the PEBL

attribute that allows specifying to which of the quality (sub-)characteristics this capability refers to. The name of the language usually refers to either BPMN or BPEL, but as there can be other workflow languages in the future, a string is used instead of a more specific enum type. All of the five elements in the feature tree can specify metrics. A metric refers to a metric type that has a name, an ID, a description, a data type (either boolean, double, long or string) and a unit (e.g., gigabyte, count, or milliseconds). The data type string should be used for more complex data types as well. A metric type can also contain a Groovy script. This is necessary for the metrics which cannot be directly measured through the observation of the engine but have to be computed from previously measured metrics. This allows that a business analyst can define the metrics and the features he requires, but a developer can then refine this, adding the correct Groovy script to compute the metrics based on measurements of other raw metrics.

Feature Tree Levels

The feature tree is a tree with a fixed set of five levels. By design, it cannot be extended infinitely as one would expect from a normal tree data structure. This is explicitly done to reduce complexity in both the DSL and the corresponding implementation. In the evaluation in Section 4.5, all benchmarks fit perfectly well into this five-level design. Hence, the downside of not being flexible enough regarding the grouping of features is not a practical issue in benchmark construction.

Test

The UML class diagram in Figure 4.3 shows the classes available to describe a test and its links to the classes of the feature tree. A test describes steps to determine metrics for a given feature declaratively. Hence, a test refers to a single feature and one or more metrics (e.g., the start of the execution as a Unix timestamp, the duration of the test execution in milliseconds, or the number of successful test cases). The test itself has a name, an ID, and a description. It links to the file containing the standard-based and engine-independent process model, and other complementary files. The test environment comprises both, the test partners and the test cases with their test steps which contain test assertions. If those elements are not expressive enough, there is the possibility to add additional data to a key-value store that is attached to any element in this package.

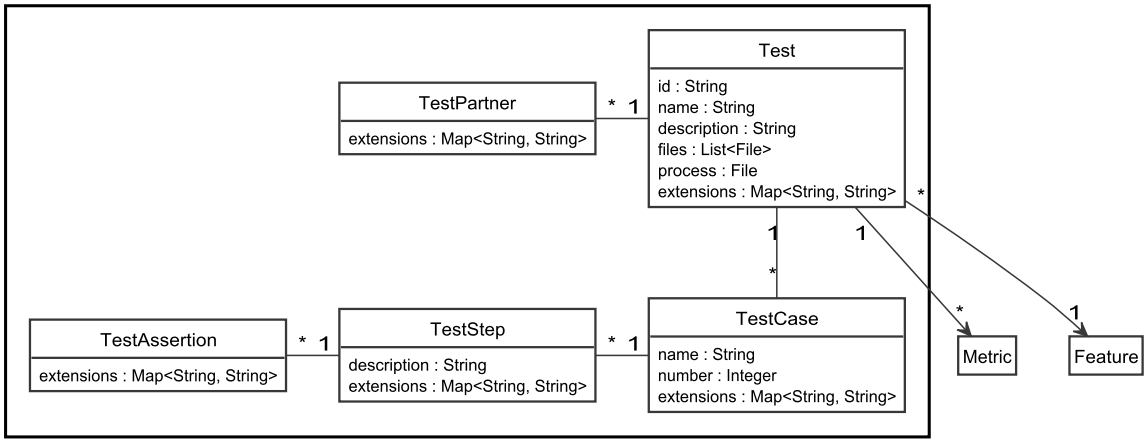


Figure 4.3.: The Tests of the Benchmark Specification of the PEBL

Domain-Specifics

At first, the UML diagram in Figure 4.3 does not look domain-specific as its structure is similar to that of typical testing languages or frameworks containing tests, test cases, steps, assertions, and test partners. The domain-specific part, however, of this testing language is encoded within the different variants for the test partners, test steps, and test assertions. Each of them is detailed in the following.

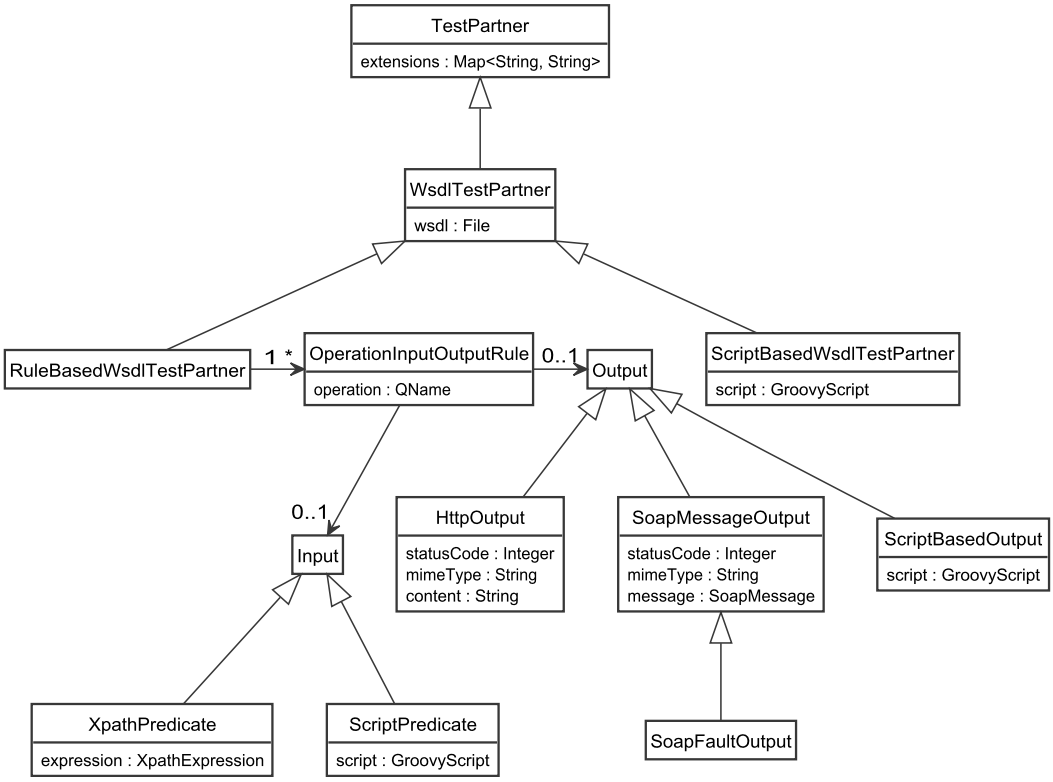


Figure 4.4.: The Domain-Specific Test Partners of PEBL

Test Partner

The language supports two different variants of WSDL-based test partners as shown in Figure 4.4: a script-based WSDL test partner and a rule-based

WSDL test partner. Both, the script and the rule-based test partner will be set up automatically by the benchmark framework because their internal logic is defined explicitly. For any WSDL test partner, the path to the WSDL file needs to be specified. The script-based one only needs the corresponding Groovy script. In contrast, the rule-based one consists of one or more rules which define the triple of operation, input, and output – with the input and output being optional. Depending on the invoked operation and the state of the input, a specific rule is fired and the corresponding output will be returned. One can explicitly define what SOAP message one expects to receive at the test partner through XPath predicates and set up a specific SOAP/HTTP message response, or even a SOAP fault. If this does not suffice, the input can be checked through a Groovy script (see *ScriptPredicate*), and the output can be computed through a Groovy script as well (see *ScriptBasedOutput*). If no input is given, any arriving message is accepted, and if no output is specified, no reply is sent. As not every test requires a test partner, a test partner can be left out as well.

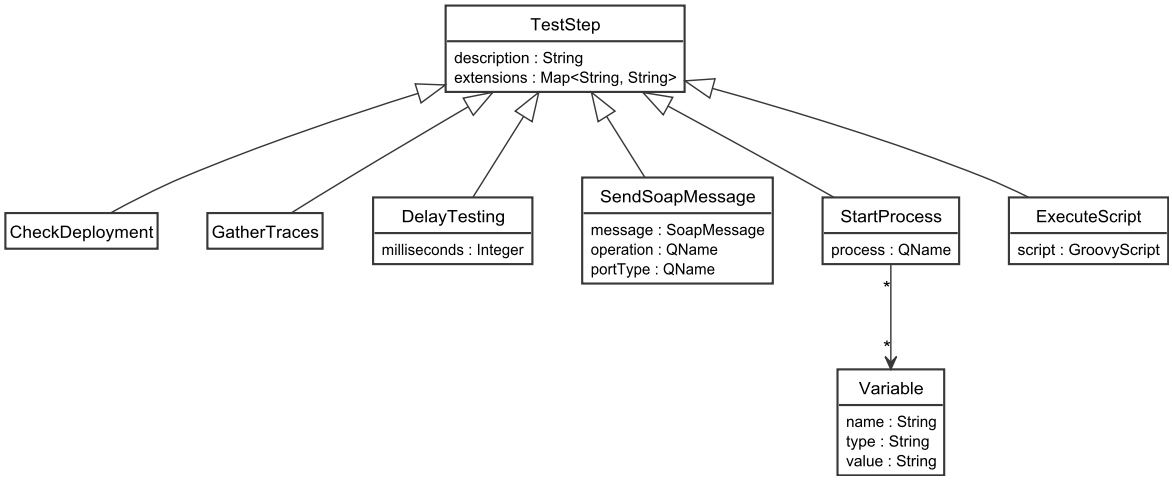


Figure 4.5.: The Domain-Specific Test Steps of PEBL

Test Steps and Assertions

PEBL contains six different test steps available as shown in Figure 4.5. Their names follow the typical naming scheme of a verb followed by an object, and they describe steps or actions that must be performed during a test case: *check deployment*, *delay testing*, *send SOAP message*, *start process*, *gather traces*, and *execute script*. Although the first five steps are domain-specific, the step *execute script* is available so that arbitrary test steps can be described. In Figure 4.6, the six different assertions are presented. Each of them asserts a specific condition after the test step it is associated to has been completed. Similarly to the test steps, there is also the *assert script* assertion which handles custom assertions. An overview of both, the different test steps and assertions with a description can be found in Table 4.1.

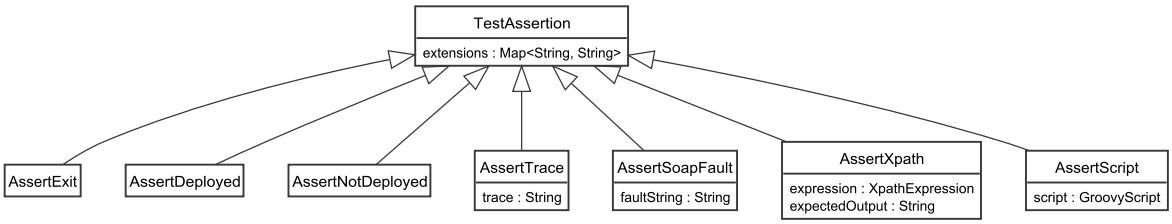


Figure 4.6.: The Domain-Specific Test Assertions of PEBL

Table 4.1.: The Domain-Specific Test Steps and Assertions of PEBL

Step	**Description**
check deployment	Determines whether the process is deployed on the engine or not.
delay testing	Holds the test execution for a given amount of milliseconds.
send SOAP message	Sends a SOAP message to an operation of a specified WSDL service and optionally receive a reply as well.
start process	Triggers the start of a process by sending variables defined by a name, type and value.
gather traces	Gathers all available execution traces from logs and engine APIs.
execute script	Describes a custom test step that acts as an extension point if the other test steps do not suffice. It will execute a given Groovy script.
Assertion	**Description**
assert exit	Asserts that the process instance has terminated.
assert deployed	Asserts that the process model is deployed. This requires that the deployment has been checked before through the *check deployment* test step.
assert not deployed	Asserts that the process model is not deployed. This requires that the deployment has been checked before through the *check deployment* test step.
assert trace	Asserts that the given trace is part of the execution trace.
assert SOAP fault	Asserts that the response of a *send SOAP message* step returned a fault with a specific fault string.
assert xpath	Asserts that the response of a *send SOAP message* step returned the expected output by evaluating it through an XPath expression.
assert script	Describes a custom test assertion that acts as an extension point if the other test assertions do not suffice. It will execute a given Groovy script. The assertion fails if an unhandled exception is raised.

4.3.2. Benchmark Results

There are three entities (i.e., *engine*, *test result*, and *aggregated result*) to describe benchmark results in PEBL. Although the engine entity can stand alone, both the test result and the aggregated result depend on the previously defined tests

and feature entities. Hence, the engine is described first, followed by the test result, with the aggregated result described last.

Engine

Within PEBL, the system under test is a process engine. Each has a name and a version (e.g., Apache ODE v1.3.6) as shown as part of Figure 4.7. Also, each engine can interpret a specific process language such as BPEL or BPMN, which has to be specified as well. Last, one can configure such engines, for instance, stating that they have to use in-memory persistence. The ID is computed based on all these information. Hence, an engine with a different configuration is another engine in our terminology. Additional information such as licensing information, URLs, or the release dates can be stored as extensions as well.

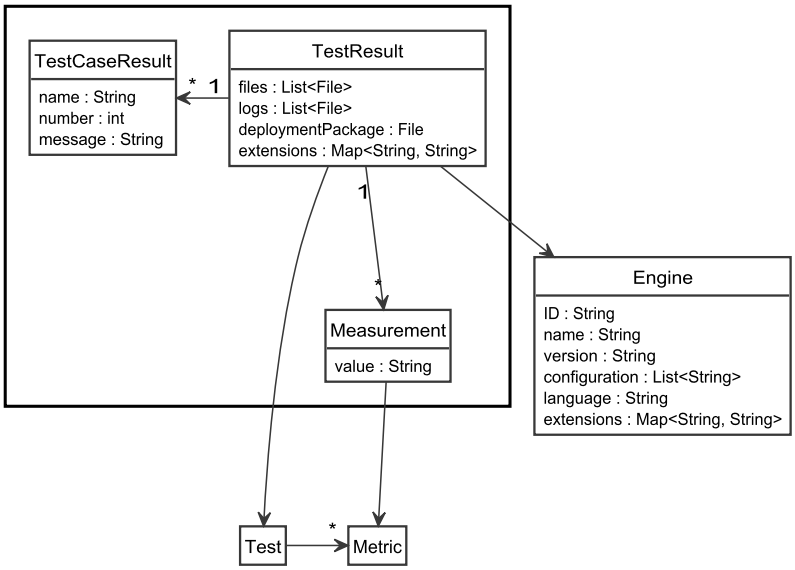


Figure 4.7.: The Test Result and the Engine in PEBL

Test Result

The raw results are encoded as test results. In Figure 4.7, one can see that the test results provide results for a given engine and test. Moreover, the tool that is used to compute the test results can be given as well. The results comprise measurements of the metrics defined by the referenced tests. Each measurement provides a value for its corresponding metric as a string. Furthermore, a test result references engine-dependent files, logs, and the deployment package containing the executed process model along with its deployment descriptors. To get insight into the result of each test case, the test case results are available as well, providing a message if they did not succeed.

Aggregated Result

As shown in Figure 4.8, the aggregated result comprises measurements scoped for a particular engine. Each measurement contains a value for a metric that can be measured via an attached Groovy script. Moreover, the metric of such a measurement is defined by an element of the feature tree. The script is used to compute the value using the available test results. Hence, instead of replicating the tree-like data structure of the feature itself, an aggregated

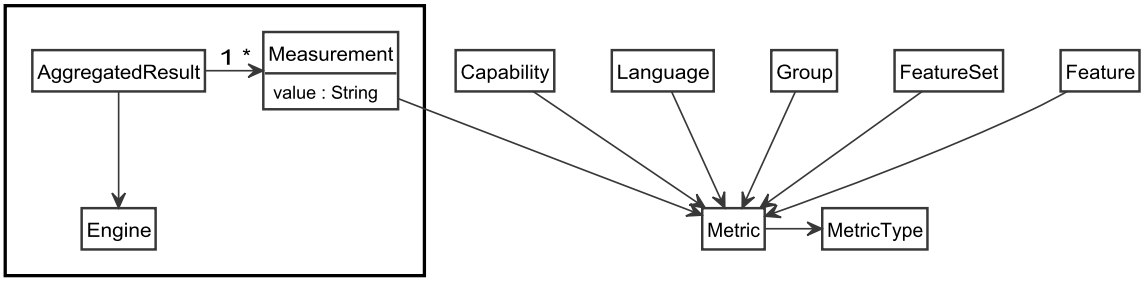


Figure 4.8.: The Aggregated Result in PEBL

result only references a metric, resulting in a flat structure within the DSL and its model.

4.4. Prototype

The prototype is written in Java 8 and makes use of an object to XML mapping specification named JAXB [34] and its implementation EclipseLink MOXy[50]. This software uses EclipseLink MOXy 2.5.2 which in turn implements JAXB 2.2. The functionality of the prototype consists of three components: 1) read and write serializations (either JSON or XML) of PEBL, 2) compute the aggregated results, and 3) validate any instance of PEBL for consistency and faults.

Architecture

The architecture of the prototype is shown in Figure 4.9. The prototype is structured into three modules according to its three functions: the *mapping*, the *aggregation*, and the *validation*. Within the mapping, specific annotations from both, JAXB and EclipseLink MOXy, are used to read and write JSON and XML instances of PEBL through EclipseLink MOXy. As part of the reading and writing, any relative paths are converted to absolute ones in memory, and when it is serialized again, the paths are relativized again. The other part, being the aggregation module, depends upon the mapping. It can recalculate the aggregated results by aggregating the available test results. Any previously aggregated result is replaced by a newly computed one. The computation itself is hard coded and based on the metric names. In the validation module, checks are available to determine the integrity and consistency of an instance of PEBL, e.g., whether the BPEL process models conform to the XSD of the standard, the WSDL files to their XSD, and the XSD files to the meta XSD. Also, these files are checked through static analysis with the tools BPELlint [101] and BPMNspector [86]. The BPELlint tool was also developed as part of this work although it is not detailed within this work.

Groovy Scripts

The aggregated results are computed by evaluating Groovy scripts. These scripts have to implement a specific interface to be able to work with this prototype. The UML diagram of this interface is given in Figure 4.10. Based on

[50] `http://www.eclipse.org/eclipselink/#moxy`, visited 2017-3-31

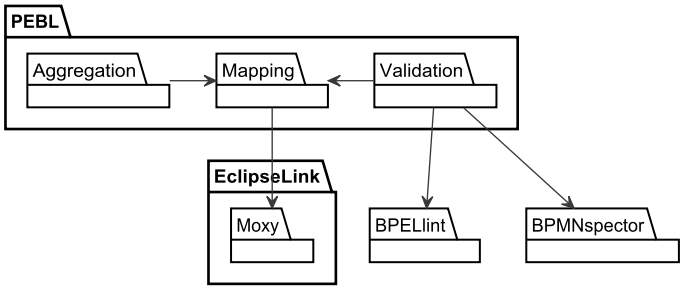


Figure 4.9.: The Architecture of the PEBL Prototype

the list of test results, the result has to be aggregated in the form of a string. Both, the *execute script* test step and the *assert script* test assertion are based on a Groovy script as well. They, however, work on the test case they are part of. Moreover, in case the assertion fails, an assertion error has to be raised.

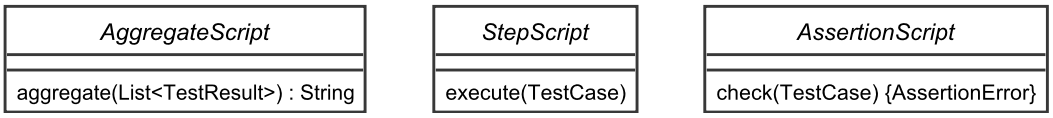


Figure 4.10.: Mandatory Interface for User-Defined Groovy Scripts in PEBL

Limitations

The prototype comes with some limitations. First, it does not support the aggregation of test results to aggregated results through user-defined Groovy scripts. The scripts are simply ignored. The prototype, does, however, contain predefined and hard-coded aggregation logic which relies on the names of the metrics. This logic already builds upon the interface described above, but the runtime interpretation of Groovy in that part has not yet been implemented. It should, however, be straightforward with the use of GroovyShell[51] as it can evaluate Groovy scripts while accessing Java objects. The same holds for the *execute script* test step and the *assert script* test assertion as well. Second, there is no GUI available to edit instances of PEBL in a user-friendly way. One can, however, edit those serializations using an XML or JSON editor, and validate them against their schemas. For more validations, the validation module of PEBL could be incorporated into a GUI as well.

4.5. Evaluation

Structure

In this section, the Process Engine Benchmark Language (PEBL) is evaluated according to the evaluation method described in Section 4.5.1. Using this method, benchmarks are crafted to determine a variety of quality characteristics

[51] `http://docs.groovy-lang.org/latest/html/api/groovy/lang/GroovyShell.html`, visited 2017-3-31

for BPEL engines in Section 4.5.2 and BPMN engines in Section 4.5.3. How the results of a benchmark are expressed is evaluated in Section 4.5.4.

4.5.1. Method

Suitability

Evaluating PEBL means to determine whether this language is suitable to express benchmarks and their results for a variety of quality characteristics for different process languages. Consequently, a large set of such benchmarks is created systematically as part of this evaluation by following the developed methods Capability to Feature Method (C2FM) and the Feature to Test Method (F2TM) subsequently. The C2FM provides guidelines on how to decompose capabilities into testable features and the F2TM provides guidelines on how to create a test for a testable feature. The suitability is determined by checking whether the DSL-elements of PEBL did suffice to express these benchmarks and how much of the extension elements as well as generic user-defined steps and assertions are required.

Quality Characteristic [113]	**Engine Capability**	**BPEL**	**BPMN**
Functional Suitability	Conformance	✓	✓
Functional Suitability	Expressiveness	✓	✓
Usability	Static Analysis	✓	✓
Resilience	Robustness	✓	
Performance Efficiency	Performance		✓

Table 4.2.: BPEL Evaluation Approach for Four out of the Eight Quality Characteristics of the ISO/IEC 25010 Quality Model [113]

Bench-marks

The benchmarks that are used for this evaluation are shown in Table 4.2. Each check mark in the table refers to a single benchmark. These benchmarks are structured according to two categories: different quality characteristics and different workflow languages. The more workflow languages and quality characteristics PEBL can cover with its DSL-elements, the more powerful and expressive PEBL is. For that reason, both BPEL and BPMN are included as process languages because they are the two most prominent workflow languages at this point (see Section 2.2.2). This evaluation focuses on four different quality characteristics, being functional suitability, usability, resilience, and performance efficiency. These four cover half of the available quality characteristics, and are crucial aspects of an engine selection decision. Hence, together, these benchmarks are sufficient to show that PEBL is general enough to cover a variety of different quality characteristics. A benchmark for such a quality characteristic is created by focusing on a capability (i.e., a part of the quality characteristic). The capabilities conformance (i.e., feature conformance), expressiveness (i.e., engine expressiveness), and static analysis (i.e., static analysis conformance) have already been defined and motivated within Chapter 2 whereas the other

two are defined and motivated in the benchmark description below. A capability is split into testable features through C2FM in Section 4.5.1.1. Thereafter, these created features are converted to tests using F2TM in Section 4.5.1.2. The general method for this whole hierarchic decomposition is given in the following, and applications of it are presented later in Section 4.5.2 and Section 4.5.3.

4.5.1.1. Capability to Feature Method

The Capability to Feature Method (C2FM) consists of two parts: *guidelines* on how to decompose abstract capabilities into specific and testable features, and *aggregation scripts* for computing aggregated results based on expected test results. C2FM

An overview of how the quality characteristics and their corresponding engines capabilities are decomposed is given in Table 4.3. The table shows the schema of the feature tree, with capability-specific names for each feature tree entity (i.e., from the group down to the specific feature). The terminology does fit for both process languages in question. Typically, it is straightforward to decompose a capability into groups, feature sets, and features. The only exception is that of performance, which only specifies experiments as features that are not grouped in any way. This is because performance results are hard to aggregate as they heavily depend upon the environment. Each decomposition is detailed later within Section 4.5. Decomposition

Quality Characteristic	Capability	Group	Feature Set	Feature
Functional Suitability	Conformance	construct group	construct	construct config.
Functional Suitability	Expressiveness	pattern catalog	pattern	pattern impl.
Usability	Static Analysis	rule set	rule	rule config.
Resilience	Robustness	fault scenario	message layer	mutation
Performance Efficiency	Performance	-	-	experiment

Table 4.3.: Capability specific names for group, feature set, and feature

As shown in Table 4.3, each capability uses its name for a feature, feature set, and group. To account for that in the serialization, these names are specified through extensions. In the case of expressiveness, for instance, the capability comprises three extension elements: *feature* set to *pattern implementation*, *featureSet* set to *pattern*, and *group* set to *pattern catalog*. With these names instead of the abstract terms, a more informative UI can be provided, as it is done for PEBDASH in Chapter 6. Extension

There are two orthogonal decomposition guidelines: 1) use existing hierarchies, and 2) start with feature sets. For almost all capabilities for which benchmarks are created, a natural hierarchy already exists (e.g., within the specification of the process language). Such an existing hierarchy is well suited to be used for the feature tree, especially to define the groups, and sometimes the feature sets as well. The second guideline is to start with the feature Decomposition Guidelines

sets. From there, it is possible to refine towards features and to abstract and categorize towards the groups. In almost any benchmark, the user is most interested in the feature set level results, as the feature level is too detailed, and the group level too abstract. Furthermore, the feature set level is typically already specified, whereas it takes effort to decompose a feature set into its testable features. For instance, to come up with rule configurations for the static analysis rules in Section 4.5.2.3, or the construct configurations for the constructs in Section 4.5.2.1 and Section 4.5.3.1.

Metrics

Each element in the feature tree can define metrics. Although there are lots of different types for these metrics, a ternary result is typical for representing success ($+$), failure ($-$), and a state in between ($+/-$) that mostly refers to a partial success (or a partial failure of one is a pessimist instead). Moreover, ternary representation is the way to express workflow pattern support [257]. A pattern is either *directly*, *partially*, or *not directly* supported.

Listing 4.1: Predefined Aggregation Algorithms

```
1 def trivalent-aggregation:
2     if only + then +
3     else if only - then -
4     else +/-
5
6 def best-value:
7     if any + then +
8     else if only - then -
9     else +/-
```

Aggregation Scripts

Because the ternary form for results is often used, two aggregation scripts are provided to help in defining the aggregation logic: *trivalent-aggregation* and *best-value*. These scripts can be seen as building blocks for more complex aggregations. They are used multiple times in the benchmarks created as part of Section 4.5. Their pseudo code is given in Listing 4.1. The algorithm trivalent-aggregation (line 1 to 4) is the default one as it shows whether all the values are either $+$ or $-$, and if this is not the case, a $+/-$ is shown. It is a clear indicator that can easily show no or full support, but it is hard to know what the $+/-$ means as it does not say whether the support is tending more towards full or no support. The algorithm best-value (line 6 to 9) is typically applied when a feature may be implemented in different ways. To know whether the feature can be used, we only need to know whether at least any of its variants is working correctly. A variant of a feature can also be a workaround that is not a perfect implementation. In that case, this variant may only provide a feature support of $+/-$ instead of $+$.

4.5.1.2. Feature to Test Method

F2TM

After the features are determined, they are converted to tests through the Feature to Test Method (F2TM). The procedure for that is depicted in Figure 4.11. It requires a feature and a process stub as an input and produces a test as its output. The process stub is important as it contains facilities to

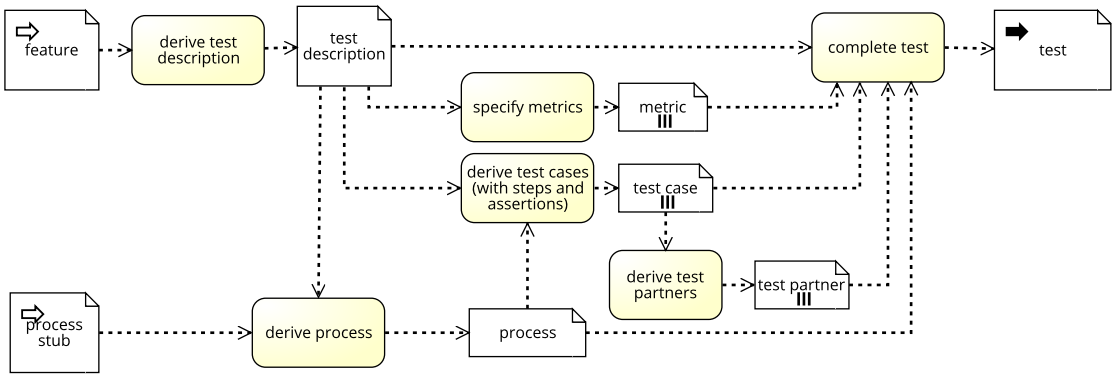


Figure 4.11.: The Feature to Test Method as a Data Flow BPMN Diagram

inject input and observe the state of instances of this process and, therefore, makes a feature under test testable. The procedure works as follows. First, the developer derives a test description from the feature, which is the basis for the remaining tasks. From that test description, metrics are defined that are to be measured. Moreover, based on the test description, the process stub is extended to provide a process that contains the feature under test, for which test cases and possibly required test partners are created. The test is complete after the metrics, process, test cases, and test partners are defined.

Process Model Stub

Creating a process model from a process model stub is suggested in Figure 4.11. This can be done in two different ways: Stub Extension (P3) and Mutated Existing Test (P4). In the first variant, a process model stub is created beforehand and then extended at special extension points. The process model stub has already set up facilities on how to inject input and observe state changes and output. In between, there are extension points where the feature under test can be put. In contrast, the Mutated Existing Test (P4) uses a fully specified process model and modifies it. This has the advantage that there is a relationship between the existing and the modified process model. If the existing test is fulfilled, and the modified one is not, it can be said that the modification introduced an issue. By keeping the difference between the existing and the modified process model to a minimum, it is simpler to evaluate whether the modification is correct in isolation. In the following evaluations, both are used depending on the specific benchmark.

Metrics

The metrics of the feature tree determine the minimal set of raw or atomic metrics that are required to aggregate the measurements of the latter to measurements of the former. Moreover, a standard set of metrics is specified, even if the measurements of these metrics stay unaggregated. This set of metrics includes metrics such as the time when the test started and its duration. An overview of the standard metrics that are typically available is given in Listing 4.2. These normally suffice for the majority of the aggregated metrics which mostly rely on the *testSuccessful* or the *testDeployable* test metrics.

Listing 4.2: Standard Test Metrics

```
<metricType dataType="long"    id="executionDuration"  unit="milliseconds"/>
<metricType dataType="long"    id="executionTimestamp" unit="timestamp"   />
<metricType dataType="long"    id="testCases"          unit="quantity"    />
<metricType dataType="long"    id="testCaseSuccesses"  unit="quantity"    />
<metricType dataType="long"    id="testCaseFailures"   unit="quantity"    />
<metricType dataType="boolean" id="testDeployable"     unit="boolean"     />
<metricType dataType="boolean" id="testSuccessful"     unit="boolean"     />
<metricType dataType="string"  id="testResult"         unit="trivalent"   />
```

Process & Test Cases & Test Partners

Making a feature testable requires the usage of existing testing methods from simple techniques such as creating equivalence classes and ensuring that coverage criteria are met up to more sophisticated methods such as combinatorial testing. They are all required to ensure that the tests test the correct feature and that the tests are complete in a way that the test result reveals how well a feature is working.

Test Validity

Also, there is the practical aspect that while having ensured that the tests are sufficient we also have to verify that the tests themselves are correct (i.e., that they can be executed successfully). This requires that the artifacts (e.g., the process model, the test cases, and the test partner definitions) are valid. The validation module of PEBL is used to find some errors quickly automatically (Automatic Static Analysis (P7)). Furthermore, by open sourcing it (Open Sourcing (P5)) and asking experts such as developers and maintainers of process engines or researchers in this field for a review (Expert Review (P6)), the quality of the tests are ensured.

4.5.2. BPEL-based Benchmarks

Dependencies

As part of this evaluation, four different BPEL-based benchmarks have been created: conformance, expressiveness, static analysis, and robustness. These four benchmarks are related. Their dependencies are shown in Figure 4.12. The most basic benchmark that is also used within the evaluation of PEAL in Section 3.5 is the Aptitude Test (P8). This Aptitude Test (P8) lays the foundation on which the conformance benchmark builds upon. The remaining three benchmarks build upon the conformance benchmark as they reuse the available features and tests to build upon or to compare against.

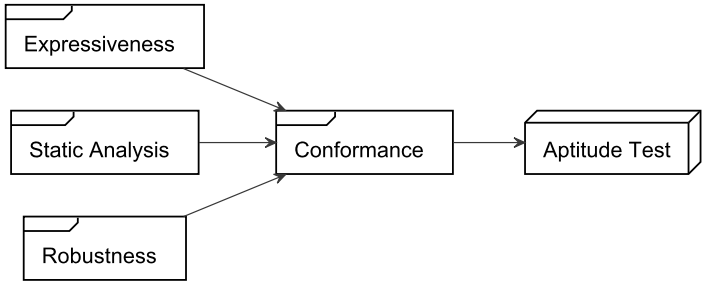


Figure 4.12.: BPEL Benchmark Dependencies

All benchmarks have in common that Message Evaluation (P12) and Partner-based Message Evaluation (P13) is used to observe the state of the process instance. This means that messages (SOAP messages in that case) are sent to the instances of the process model under test, and the response is checked to contain specific values via XPath expressions. This may require additional test partners which also can receive and send predefined or computed messages. Moreover, the state of the test partners can also be queried for further assertions such as whether a particular message has been received. An example of such a message exchange is given in Figure 4.13 for both variants with the Message Evaluation (P12) on the left-hand and the Partner-based Message Evaluation (P13) on the right-hand side.

Message Evaluation (P12)

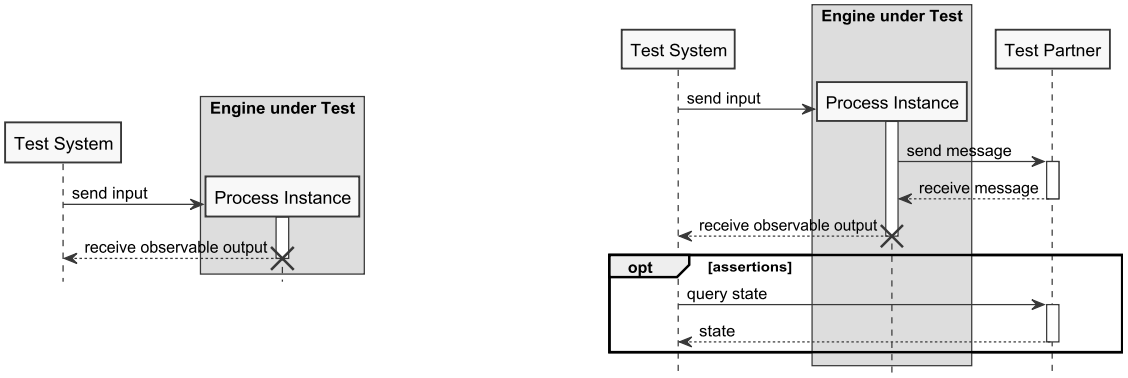


Figure 4.13.: Typical Scenario for Message Evaluation (P12) (left) and Partner-based Message Evaluation (P13) (right)

In some cases, Concurrency Detection (P16) is required as well which is built upon Partner-based Message Evaluation (P13). A test partner is simulated which can receive messages, waits for a specific amount of time, and then responds to the previously received messages. It is measured whether such request-response pairs do overlap, or, in other words, whether the test partner receives multiple calls in parallel. An example of the exchanged messages and their intents is given in Figure 4.14. It subdivides the message exchanges into three subsequent sections according to the paradigm *arrange-act-assert*. First, the test partner is informed that he has to detect concurrency (i.e., arrange). Next, the messages are exchanged (i.e., act), and last, the test system queries the test partner whether concurrency has been detected (i.e., assert). Of course, this method cannot distinguish between real concurrency or the simulation of concurrency through nondeterministic interleaving.

Concurrency Detection (P16)

In the following, all four BPEL-based benchmarks are presented in detail.

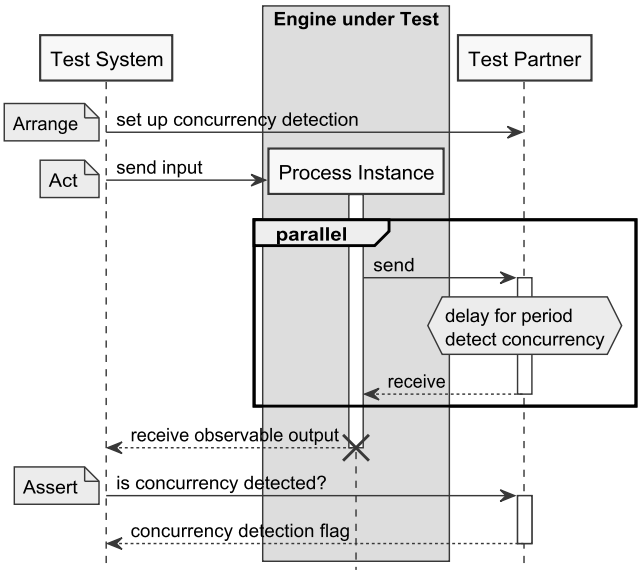


Figure 4.14.: Typical Scenario for Concurrency Detection (P16) using Partner-based Message Evaluation (P13)

4.5.2.1. BPEL Conformance Benchmark

Parts of this section have been taken from [95].

The first benchmark has been created to evaluate the capability feature conformance, or conformance in short, for the language BPEL. It is expressed in PEBL in the following. The constructs of BPEL (e.g., activities and scopes) are the feature sets. Those feature sets are organized into construct groups, and each construct configuration represents a testable feature. The benchmark for this capability evaluates the quality characteristic functional suitability, or to be more precise, the two sub-characteristics functional completeness and functional correctness.

C2FM **Capability to Feature Method for Conformance** The features are derived from the requirements defined in the BPEL specification [181] using the notational conventions [112] (e.g., MUST, MUST NOT, or REQUIRED) as they define what the capability conformance means for BPEL. Specifically, features for every activity, attribute, and fault that is part of executable BPEL are provided. To ensure that all features of BPEL are covered, Configuration Permutation (P1) is applied to determine that each configuration of a construct is captured in a feature. For this, the existing hierarchic structure of the BPEL specification is used. It comprises basic activities, structured activities, and scopes, which, in turn, group constructs that can be configured in various ways. An overview of these three groups with their constructs and the number of construct configurations is given in Table 4.4. In the following, these groups and their constructs are explained in more detail.

Table 4.4.: BPEL Conformance: Config. per Construct within Construct Group

basic activities	78	**structured activities**	48	**scope**	43
Assign	27	Flow	12	Compensation	6
Empty	1	ForEach	14	CorrelationSets	2
Exit	1	If	6	EventHandlers	21
Invoke	18	Pick	16	FaultHandlers	13
Receive	5	RepeatUntil	3	MessageExchanges	4
ReceiveReply	14	Sequence	1	PartnerLinks	1
Rethrow	3	While	2	Scope-Attributes	3
Throw	5			TerminationHandlers	3
Validate	2			Variables	2
Variables	3				
Wait	3				

Groups & Constructs

The *basic activities* group contains constructs for every basic activity of BPEL [181, pp. 84–97]. This includes the *invoke*, *receive*, *reply*, *assign*, *throw*, *wait*, *empty*, *exit*, *validate*, and *rethrow* activities, as well as faults related to them. The second group, *structured activities*, comprises structured activities [181, pp. 98–114]. This includes *sequence*, *if*, *while*, *repeatUntil*, *pick*, *flow*, and *forEach* activities. Again, faults related to these activities belong to this group as well. Although being structured, *scopes* [181, pp. 115-147] are treated separately. The last group contains constructs for *scopes*, *fault-*, *compensation-*, *termination-*, and *eventHandlers*. Furthermore, the scope-local definition of *variables*, *partnerLinks*, *messageExchanges* and *correlationSets* is also investigated here.

Example

For instance, the *if* activity of BPEL is a construct which is part of the *structured activities* group and has six configurations in total. The four functional configurations, namely, 1) an if without any additional elements (*If*), 2) an if with an else (*If-Else*), 3) an if with an else if (*If-ElseIf*), and 4) an if with an else if and an else (*If-ElseIf-Else*) refer to the good cases when the condition expression is valid. In contrast, the other two configurations refer to the handling of invalid condition expressions and specify which fault is required to be thrown in each case.

Limitations: Unspecified Language Features

Three areas required for executable BPEL are not completely specified and remain a design-choice for an implementer of the standard. This is the exact structure of a partner reference, necessary for the assignment of *partnerLinks*, the URI scheme used to identify XSL stylesheets and the behavior of the engine if a fault is propagated to, and not handled by, the root-level scope of a process that still has open request-response interactions. WS-Addressing *EndpointReferences* [272] (encapsulated in a *service-ref* container of BPEL) are used as partner references and to identify XSL resources by their filename. This implies that an engine that supports dynamic binding without *EndpointReferences* will fail our test case. Concerning fault propagation, we test for the mechanism applied

by most high-level programming language, such as Java or C#, which is also a prerequisite for distributed fault handling [89]: It is expected that an uncaught fault at root-level is forwarded to the recipients in open request-response operations.

Limitations: Composition

In this benchmark, the BPEL activities and their configurations are covered. What is not covered are combinations of arbitrary BPEL activities. This would lead to test explosion, as they can be nested indefinitely through the structured activities. The expressiveness evaluation in Section 4.5.2.2, however, covers combinations of BPEL activities in the form of pattern implementations. Similarly, but in a limited scope, the robustness evaluation in Section 4.5.2.4 covers two combinations of BPEL activities as well.

Extensions

For the conformance benchmark for BPEL, no extension elements are used except for specifying the benchmark-specific names for feature, feature set, and group. Extension elements, however, could be used to indicate which construct configuration is a workaround or alternative for another construct configuration. This is especially helpful if it is determined that an important feature is not supported. If, however, there are workarounds or alternatives available, this would not matter that much. Furthermore, one can specify which construct configurations are part of CoreBPEL [110] (i.e., a subset of BPEL without syntactic sugar). With that information, it would be possible to determine whether a feature is actually syntactic sugar and which feature is crucial to fulfill as other features could be mapped to that one (or seen as an alternative).

Metrics

In this benchmark, we are interested in metrics of the construct configurations, constructs, groups, and the support in general. Regarding the construct configuration, it is necessary to know whether they are deployable to the engines (*testDeployableCount* that requires *testDeployable*) and whether they work according to the specified test cases (*testSuccessfulCount* that requires *testSuccessful*). Regarding the constructs, it is necessary to know whether all, none, or only some of the configurations are supported (*testResultTrivalentAggregation* that requires *testResult* using the *trivalent-aggregation* algorithm), and how many configurations are available for each construct (*featuresSum*). Furthermore, for the construct groups and the language, the questions how many features there are (*featuresSum*) and how many of them are supported in total (*supportedFeaturesCount* that requires *testSuccessful*) are asked.

F2TM

Feature to Test Method for Conformance It is necessary to derive tests for the previously defined features. F2TM is applied as follows. Each conformance test describes a specific feature of BPEL in relative isolation. A test comprises BPEL, WSDL, XSD, and XSLT files as well as test cases and test partners. Every process model implements the same WSDL interface to enable a unified handling of the tests. The interface is intended to be as simplistic as possible to ensure that all engines support it while being complex enough to test all features

of BPEL. It contains a *partnerLinkType* and several message definitions, with all messages containing a single message part of the type integer. Thereby, we avoid problems that result from the processing of large documents, as it is not our intention to assess XML processing capabilities here. The *portType* is made up of two operations, namely (i) a synchronous one that may also reply with a fault and (ii) an asynchronous one. The binding for these operations is the most basic and plain one available, thereby having a high probability of being supported by every engine: document/literal style over HTTP [271, Sec.3]. This is the preferred binding for achieving interoperability according to WS-I Basic Profile 2.0 [292] by the Web Services-Interoperability Organization (WS-I). Additionally, every message exchanged by these operations is configured to be useable in correlation sets. A similarly structured WSDL definition is provided for a partner service that is required to test *invoke* activities.

Observability

The tests for a specific feature of BPEL are not strictly isolated, which is also no requirement for conformance tests [166, pp. 203-208]. Some features are not testable in isolation, such as *faultHandlers* that require a fault to be thrown in the first place. Furthermore, to verify the correctness of a test, it is necessary to have an output available that can be evaluated. Consequently, all process definitions we use as conformance tests contain certain elements and most do contain synchronous operations. Listing 4.3 outlines the general structure that applies to most tests. The activities therein with their specific configuration could be verified to be supported by all engines, so they do not influence the results of other tests. Here, Stub Extension (P3) using Listing 4.3 is applied. This stub is similar to the process model used in the evaluation of PEAL in Section 3.5. Hence, it has been asserted that it works on any engine, and based on that, it can be extended precisely with the feature under test. In other words, the stub has no effect on the conformance test result.

Listing 4.3: Outline of the Process Definition Stub for BPEL Process Models

```
1  <process>
2     <partnerLinks />
3     <variables />
4     <sequence>
5       <receive />
6       <!--Test implementation-->
7       <assign />
8       <reply />
9       <!--More test implementation, if message exchanges are involved-->
10    </sequence>
11 </process>
```

Invoke Example

Listing 4.4 shows an example of a serialization of a nontrivial test, namely, for the BPEL activity *invoke* which makes a synchronous call to a test partner. It closely follows the message sequence on the right-hand side of Figure 4.13. In the test case, first, it is asserted that the process model is deployed (line 7–9). Next, a SOAP message containing the number 1 is sent to the WSDL interface of the process model (line 10–17), which will send the number 1 to the test partner subsequently. This test partner is a rule-based WSDL test partner who responds to messages containing the number 1 with a message containing the

number 1 (line 22–38). The test case is successful if the received response does contain the number 1 (line 12).

Listing 4.4: PEBL Serialization of BPEL conformance test for feature *Invoke-Sync*

```
1  <test feature="Conformance__BPEL__basic__Invoke__Invoke-Sync">
2      <process>Invoke-Sync.bpel</process>
3      <description>A receive-reply pair with an intermediate synchronous invoke.</
           description>
4      <testCases>
5          <testCase name="Good-Case-1" number="1">
6              <testSteps>
7                  <checkDeployment>
8                      <testAssertions><assertDeployed /></testAssertions>
9                  </checkDeployment>
10                 <sendSoapMessage>
11                     <testAssertions>
12                         <assertXpath expression="..." expectedOutput="1" />
13                     </testAssertions>
14                     <operation>startProcessSync</operation>
15                     <portType>testInterface</portType>
16                     <message><!-- send 1 --></message>
17                 </sendSoapMessage>
18             </testSteps>
19         </testCase>
20     </testCases>
21     <files>TestInterface.wsdl TestPartner.wsdl</files>
22     <testPartners>
23         <ruleBasedWSDLTestPartner>
24             <wsdl>TestPartner.wsdl</wsdl>
25             <rules>
26                 <rule operation="startProcessSync">
27                     <xpathPredicate><!-- is 1? --></xpathPredicate>
28                     <soapMessageOutput statusCode="200"
29                                        mimetype="application/soap+xml">
30                         <!-- 1 -->
31                     </soapMessageOutput>
32                 </rule>
33             </rules>
34         </ruleBasedWSDLTestPartner>
35     </testPartners>
36     <metrics />
37 </test>
```

Test Cases & Test Partners

As shown in the previous example, the test cases and the test partners together specify the sequence of messages to be exchanged and the assertions to be made. To ensure that the activity under test is fully covered, test metrics such as branch coverage have been applied as well. For instance, the test to evaluate the *if* activity contains two test cases: one tests whether the *condition* of the *if* is evaluated to true, and the other one to false. This achieves full branch coverage for that particular test.

Parallel Activities

There are three activities that allow parallel execution in BPEL: *forEach* with the attribute *parallel* set to *true*, a *flow* with parallel branches, and *onEvents* which are executed in parallel if the scope they are attached to is active. There are two variants to evaluate those features: simply evaluate whether the activities will behave as expected from a black-box point of view, or try to determine whether the activity is using concurrency. The latter is realized through Concurrency Detection (P16) outlined before and the former is covered with typical functional testing.

Metrics

For BPEL conformance tests, the standard test metrics outlined in Section 4.5.1.2 are sufficient. No additional ones need to be specified.

4.5.2.2. BPEL Expressiveness Benchmark

Parts of this section have been taken from [96].

The second benchmark covers the capability expressiveness for the language BPEL. Similarly to the previous benchmark that covered conformance, this benchmark evaluates a different part of the quality characteristic functional suitability with its two sub-characteristics functional completeness and functional correctness.

Capability to Feature Method for Expressiveness An overview of the patterns and the amount of different pattern implementations is found in Table 4.5. For this benchmark, the already available structure of a pattern catalog with patterns that in turn may be one or more pattern implementations are reused. The patterns are the feature sets which are organized in pattern catalogs, and each pattern implementation represents a testable feature. Four patterns, namely, *WCP-08 Multi-Merge*, *WCP-09 Structured Discriminator*, *WCP-10 Arbitrary Cycles*, and *WCP-15 Multi-Instance Without A Priori Run-Time Knowledge*, do not have any pattern implementations, as their language support already states that they are not directly supported because of the inherent characteristics of BPEL [149]. Although it is possible to implement anything using a Turing-complete language [255], in these cases, it would require too many BPEL activities to implement these patterns. For the three patterns *WCP-17 Interleaved Parallel Routing*, *WCP-18 Milestone*, and *WCP-19 Cancel Activitiy*, at most partial support is possible are there is no BPEL activity available in the BPEL specification that would directly implement these patterns, but one needs to use a bunch of BPEL activities together to implement it. For the remaining patterns, there is a corresponding BPEL activity. But sometimes, a few BPEL activities together can act as a workaround to achieve the same behavior as well. In these cases, the pattern may have different pattern implementations. C2FM

For instance, the pattern *WCP04 exclusive choice* is part of the workflow control flow pattern catalog group and has a single pattern implementation making use of an *if* activity with an *else* branch. The pattern *WCP06 multi choice* has two pattern implementations, namely, one direct implementation using *flow* with *links* and one partial implementation using *flow* with *if* activities instead of *links*. Example

For the feature tree, the language support is added as an extension for both, the pattern implementation and the pattern itself. This language support defines the upper bound of the support that an engine can expect to achieve if it supports the full BPEL specification. The language support for the patterns is specified in Table 4.5. The language support for pattern implementations, however, can deviate from one of the patterns for workarounds that only provide partial support ($+/-$) for a pattern with direct language support ($+$). Extensions

Table 4.5.: BPEL Expressiveness: Pattern Catalog, Patterns, and Number of Pattern Implementations

Control-Flow Pattern Catalog [257]		**BPEL**	24
WCP-01	Sequence	+	1
WCP-02	Parallel Split	+	1
WCP-03	Synchronization	+	1
WCP-04	Exclusive Choice	+	1
WCP-05	Simple Merge	+	1
WCP-06	Multi-Choice	+	2
WCP-07	Structured Synchronizing Merge	+	2
WCP-08	Multi-Merge	-	0
WCP-09	Structured Discriminator	-	0
WCP-10	Arbitrary Cycles	-	0
WCP-11	Implicit Termination	+	1
WCP-12	MI Without Synchronization	+	6
WCP-13	MI With A Priori Design-Time Knowledge	+	2
WCP-14	MI With A Priori Run-Time Knowledge	+	1
WCP-15	MI Without A Priori Run-Time Knowledge	-	0
WCP-16	Deferred Choice	+	1
WCP-17	Interleaved Parallel Routing	+/-	1
WCP-18	Milestone	+/-	1
WCP-19	Cancel Activity	+/-	1
WCP-20	Cancel Case	+	1

Metrics The metric that is relevant is the actual *support* of the pattern and pattern implementation. This means that if the test is successful (*testSuccessful*), the previously specified language support is the actual support of that pattern implementation, which can either be direct ($+$) or partial ($+/-$) support. Otherwise, the pattern implementation is not directly supported ($-$). The support of a pattern is the best available support of its pattern implementations. In that case, the algorithm best-value is applied. Another metric is whether an engine fulfills the language support of the pattern. This is computed by comparing the language support with the actual support of a pattern (*standardFulfilled*). On top, the amount of patterns that fulfilled the standard can be computed as well (*standardFulfilledCount*).

F2TM **Feature to Test Method for Expressiveness** The actual BPEL process models for the pattern implementations are from Lenhard [147] and Lenhard et al. [149]. Their pattern implementations are incorporated into PEBL by reusing the process stub, its WSDL interfaces, and test partners from the BPEL conformance benchmark, effectively applying Stub Extension (P3). Hence, the actual serialization of such a test is omitted as it does not differ really from the conformance example that has been shown previously. To compute the aggregated metrics, the standard test metrics are sufficient. Merely the *testSuccessful* metric is required, as all aggregated metrics depend only upon this one.

The major difference, however, is that instead of testing a single BPEL activity in isolation, this benchmark evaluates also multiple BPEL activities working together to fulfill a particular pattern.

4.5.2.3. BPEL Static Analysis Benchmark

Parts of this section have been taken from [99].

The third benchmark covers the capability static analysis for the language BPEL. That capability corresponds to the quality characteristic usability, or, to be more precise, the sub-characteristic user error protection. This benchmark builds upon the conformance benchmark from Section 4.5.2.1 conceptually but also feature- and test-wise.

Tags

The BPEL [181] specification contains 94 static analysis rules. Because of the large number of rules, they are tagged according to two dimensions: violation check (*How* are the targets checked?) and target elements (*What* BPEL activities and elements are restricted further?). These tags allow gaining additional insight as they can be used to group results in a comprehensive and easy to interpret way. In the following, the tags are outlined that apply to the covered[52] static analysis rules from Table 4.6. An overview of the tags and the rules is presented in Table B.1. A rule is tagged at least one time per dimension, and can be tagged multiple times within each tag dimension.

Dimension Violation Check

The *violation check* tags describe the type of the check. The three tags with the highest number of rules, namely, *node requirement*, *choice*, and *uniqueness* compensate the lax XSD of BPEL. The rules tagged with *node requirement* requires specific elements or attribute values. In other words, BPEL elements are restricted further because the BPEL schema offers a greater choice which could result in wrong element or attribute configurations. If a rule demands the usage of attributes and elements in specific combinations, it is tagged with *choice*. In this case, *choice* can refer to an *inclusive or* or an *exclusive or*. A more strict XSD could have made these rules obsolete in the first place by using, for example, the native XSD choice mechanism instead. Rules with the *uniqueness* tag check the uniqueness of attributes or elements. For instance, the name attributes of `<variable>` definitions have to be unique per `<scope>`. The native XSD mechanisms for uniqueness could have rendered these rules redundant. Several rules are tagged with *consistent redundancy* that deal with nodes which carry redundant information that can be derived from the context (e.g., from other attributes, elements, or the location of an activity). The *location* tag is assigned to rules that restrict possible parents or ancestors of activities. Such rules are necessary due to undesired inheritance defects of the BPEL XSD types. For instance, `<rethrow>` is a common activity even though it is solely applicable in the context of `<faultHandlers>`. The rules with the tag *execution instructions*

[52] A complete tagging of all rules can be found in the accompanying technical report [202].

instruct the BPEL engine how to execute the process (e.g., how to lookup a variable at runtime). Static analysis can detect errors that occur when the execution would be performed in the correct order, but invalid execution at runtime remains unchecked by static analysis, of course. Hence, these rules can only be checked up to a certain point with static analysis. Rules are tagged with *definition resolution* if the rules require that a BPEL activity references other BPEL, WSDL or XSD elements by a Qualified Name (QName) or other references. Control cycles are forbidden in BPEL and the rules that describe their detection are tagged with *control cycle detection*.

Dimension Target Elements

The tags in the *target elements* dimension indicate which BPEL activities or elements are restricted further. The majority of the rules refer to activities related to the message exchanges and their required WSDL and XSD definitions, whereas only a minority of the rules restricts structured activities.

Big Picture

The big picture of the benchmark creation is shown in Figure 4.15. It reveals that the static analysis benchmark builds upon the conformance benchmark from Section 4.5.2.1 on the left-hand side. On the right-hand side, the creation of the static analysis benchmark is pictured which consists of two steps: first, the formalization of the static analysis rules (i.e., C2FM), and, thereafter, the mutation of existing conformance tests through fault injection to get to the static analysis tests (i.e., F2TM). Furthermore, two different types of metrics are proposed to measure the fulfillment of any rule configuration: the *classical result* (i.e., the result of the static analysis test), and the *pairwise result* (i.e., the result of the static analysis test combined with the result of the corresponding conformance test). All parts of the big picture are detailed in the following, starting with the formalization.

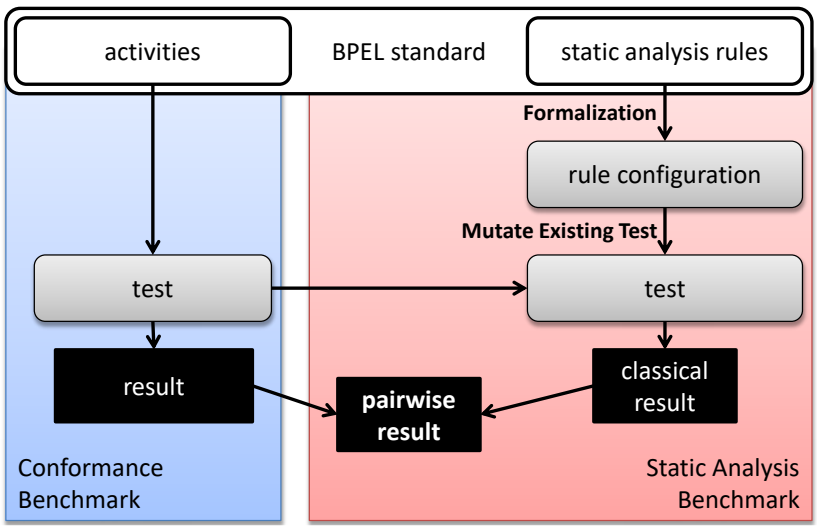


Figure 4.15.: BPEL Static Analysis: Big Picture

C2FM

Capability to Feature Method for Static Analysis The *static analysis rules* are the feature sets which are organized in *rule sets*, and each *rule configuration*

represents a testable feature. In this case, there is only a single rule set with 94 static analysis rules.

Formalization

The first step, according to the big picture, is to formalize the static analysis rules of the BPEL specification. These formalizations are either taken and adapted from Kopp et al. [134], or have been newly created [202]. The formalization of the rules can be found in the accompanying technical report [202]. With these formalizations, it is possible to derive permutations that contain all the valid and invalid combinations of the relevant BPEL elements for this rule. The formalization is required to ensure that the resulting tests are complete and their numbers do not explode. In this work, the existing BPEL formalization of Kopp et al. [134] is used whenever possible. As they formalized the rules positively (i.e., specifying valid combinations of BPEL features), we have to negate them for creating invalid combinations which are necessary for the fault tests. When there is no formalization available, the rule is modeled as part of this work. To ensure a high quality, peer-reviews within the Distributed Systems Group (DSG) at the University of Bamberg are conducted. Moreover, the formalizations are also compared with the those from Kopp et al. [134] to detect issues. By permuting the formalizations, a list of combinations is produced which can be subdivided into valid and invalid configurations. The invalid combinations are the rule configurations that an engine should detect and reject. Hence, precisely these are the "features" which need to be converted to tests. The valid ones are already part of the conformance benchmark, and no additional features or tests are required for them.

Example

For instance, the rule SA00001 is part of the standard-conformant static analysis rules group and has two rule configurations, namely, 1) handling the Message Exchange Pattern (MEP) notification, and 2) handling the MEP solicit response.

Rule Configurations

In this work, only a subset of all rules are covered. To be precise, 71 of the 94 rules as shown in Table 4.6 are formalized, and the remaining 23 are out of scope. The latter are not covered because they either are engine-specific and we focus on engine-independent static analysis, or make use of expression parsing which would require us to test the expression language XPath [269] as well. Rules #56 and #77 are the exception. They are not covered because the conformance benchmark lacks features that evaluate varying combinations of nesting multiple activities as this results in test explosion. And such combinations would be required as positive tests for these two rules. In total, there are 762 rule configurations. This means that, on average, a rule has nine rule configurations. However, the actual number varies greatly between the rules due to the different amount of configurations that need to be checked. Rule #3 has the most rule configurations with 342, whereas 26 rules solely require exactly one rule configuration. Another 33 rules comprise more than one but less or equal than ten rule configuration. Only eleven rules have more than ten

Table 4.6.: BPEL Static Analysis: Rules and their Rule Configurations

rule	762	**rule**		**rule**		**rule**		**rule**	
SA00001	2	SA00018	2	SA00047	12	SA00065	4	SA00084	1
SA00002	1	SA00019	2	SA00048	20	SA00066	6	SA00085	5
SA00003	342	SA00020	13	SA00050	2	SA00067	1	SA00086	2
SA00005	5	SA00022	3	SA00051	1	SA00068	1	SA00087	8
SA00006	12	SA00023	2	SA00052	1	SA00069	1	SA00088	1
SA00007	10	SA00024	1	SA00053	4	SA00070	15	SA00089	1
SA00008	10	SA00025	5	SA00054	2	SA00071	3	SA00090	2
SA00010	21	SA00032	77	SA00055	1	SA00072	2	SA00091	1
SA00011	1	SA00034	8	SA00057	12	SA00076	1	SA00092	2
SA00012	1	SA00035	1	SA00058	20	SA00078	2	SA00093	10
SA00013	2	SA00036	1	SA00059	1	SA00079	4	SA00095	1
SA00014	28	SA00037	1	SA00061	36	SA00080	2		
SA00015	4	SA00044	2	SA00062	1	SA00081	5		
SA00016	1	SA00045	2	SA00063	1	SA00082	1		
SA00017	1	SA00046	2	SA00064	1	SA00083	2		

rule configurations. In Table 4.6, the rule configurations per rule are explicitly depicted.

Metrics

The metrics for the rule configurations are the *classicalResult* and the *pairwiseResult*. The *classicalResult* is either $+$ if the test succeeds (i.e., the process model is rejected at deployment) or $-$ if the test fails (i.e., the process model is accepted at deployment). It builds directly upon the standard *testDeployable* metric. The other metric, *pairwiseResult*, is different as it builds upon both the *classicalResult* and the *testDeployable* metric of the corresponding conformance test. The computation logic is outlined in Table 4.7. By incorporating the conformance test result as well, it is possible to get a more precise picture of the actual support: only engines that understand process models can effectively reject erroneous process models. In case both process models are rejected, the underlying feature is simply unsupported, and the static analysis rule is not implemented. The erroneous process model is rejected because of the wrong reason. The pairwise metric show a much clearer image in how well the static analysis rules are implemented and supported by the engines [99].

Table 4.7.: Classical (left) vs. Pairwise (right) Result Metric Algorithm

static analysis test	
deployed	**rejected**
$-$	$+$

base conformance test	**static analysis test**	
	deployed	**rejected**
deployed	$-$	$+$
rejected	$-$	$-$

Extensions

For the static analysis, the previously described tags are stored in an extension element for the rules (i.e., feature sets) as a comma separated list.

Feature to Test Method for Static Analysis Next, the tests are derived from the rule configurations that represent invalid combinations of BPEL elements and attributes. This is done by selecting an appropriate test from the conformance benchmark that includes a valid combination of BPEL elements and modifies this correct feature test to contain the invalid combination, hence, creating a fault test. Also, the quality of the tests increases as we identify minimal required changes. F2TM

To determine which conformance test to use, the BPEL activities and attributes that are used in the formalization of the rule configuration provide the starting point. For example, the following partial formalization of the rule #47 comprises receiving a nonempty message through an `<onMessage>` activity [201]: Fault Tests Creation

$$[\texttt{@variable}, \text{none}] \times [\texttt{<fromParts>}, \text{none}]$$

To violate the rule, the received message must not be completely assigned to a variable via the `@variable` attribute nor any of its parts to variables via the `<fromParts>` and its `<fromPart>` elements. As the conformance test, the smallest process model is picked from the BPEL conformance benchmark that provides all necessary BPEL elements. In the example, this is *Pick-CreateInstance-FromParts*. The formalization shows the necessary modifications that have to be made so that the process models of a test-pair differ only minimally. In this example, the `<fromParts>` element is removed from the process model of the *Pick-CreateInstance-FromParts* conformance test, resulting in the *NoVariable-NoFromPart-OnMessage* static analysis test. In this example which applies Mutated Existing Test (P4), only a single rule is violated by this erroneous process model. This condition is important as it isolates a single rule configuration (i.e., static analysis violation). For optimal test results, each test shall violate a single rule and have a minimal difference to its feature test, however, as a few rules are not disjunct in their validation, this optimal state cannot be guaranteed for every test. From the 762 tests, only 6% violate multiple rules. Put differently, over 94% contain an isolated fault.

The standard metrics for a test suffice. In fact, the *testDeployable* metric is the only one that is necessary. The other ones can be measured along the way, but are not relevant to the metrics of the feature tree. Metrics

The tests make use of two extension elements: *staticAnalysisChecks* and *base*. These tests contain erroneous process models, and the errors in these process models can be detected through static analysis. The validation package of PEBL makes use of static analysis checks. By setting the extension element *staticAnalysisChecks* to *false*, it is indicated that the process model should not be validated and that the process models contain errors explicitly. Furthermore, a test for a static analysis feature is based on a test for a conformance feature (see Mutated Existing Test (P4)) to ensure that only the injected fault is isolated, Extensions

and to compute the pair-wise static analysis metric. To have this information available, the *base* extension element stores the *id* of the conformance test.

Example An example of the rule configuration *SA00001-Notification* of the static analysis rule *SA00001* is given in Listing 4.5. It shows the test structure that is used for all the tests in this benchmark. It solely contains a single test step that checks the deployment and asserts that the process model is *not* deployed. The test lacks any test partner and any test steps that interact with an actual instance of the process model because that is not necessary for this benchmark.

Listing 4.5: Example of the Benchmark for the rule configuration SA00001-Notification of the Static Analysis Rule SA00001 of the BPEL specification using PEBL.

```
<test feature="SA00001-Notification" id="SA00001-Notification__test">
    <process>SA00001-Notification.bpel</process>
    <testCases>
        <testCase name="Good-Case-1" number="1">
            <testSteps>
                <checkDeployment>
                    <testAssertions>
                        <assertNotDeployed/>
                    </testAssertions>
                </checkDeployment>
            </testSteps>
        </testCase>
    </testCases>
    <files>TestInterface.wsdl</files>
    <testPartners/>
    <extensions>
        <staticAnalysisChecks>false</staticAnalysisChecks>
    </extensions>
</test>
```

4.5.2.4. BPEL Robustness Benchmark

Parts of this section have been taken from [98].

The last of the four benchmarks covers the capability robustness for the language BPEL. Robustness is part of the resilience quality characteristic and its sub-characteristic fault tolerance.

Method **Capability to Feature Method for Robustness** To evaluate message robustness properties of a BPEL engine, the runtime behavior of instances of robust processes that interact with faulty partner services is observed. It is determined whether the process instances can react on the faulty response of the partner service. This message robustness is denoted as *backdoor message robustness* as the faults are injected as response of an external third party service. This is in contrast to *frontdoor message robustness* in which the faulty messages are sent directly as requests to an existing instance or creating a new one. Robust processes use fault handling and validation constructs available in the process language which would achieve fault tolerance when executed on a robust engine. Hence, we create a process for each of those constructs to evaluate the actual message robustness capabilities of each of those constructs of an engine. Also, the partner services have to be configured to respond with faults that

are suitable to reveal message robustness issues. It is proposed to create false responses systematically for each of the layers. Moreover, the false responses are created by applying one mutation to the correct response to test each fault in isolation. A test is a combination of using a robust process with a specific fault handling construct that needs to handle a specific faulty response on a specific engine. When a fault is handled by at least one robust process, the engine is considered robust regarding this faulty response. Upon this data, message robustness properties can be derived per message layer.

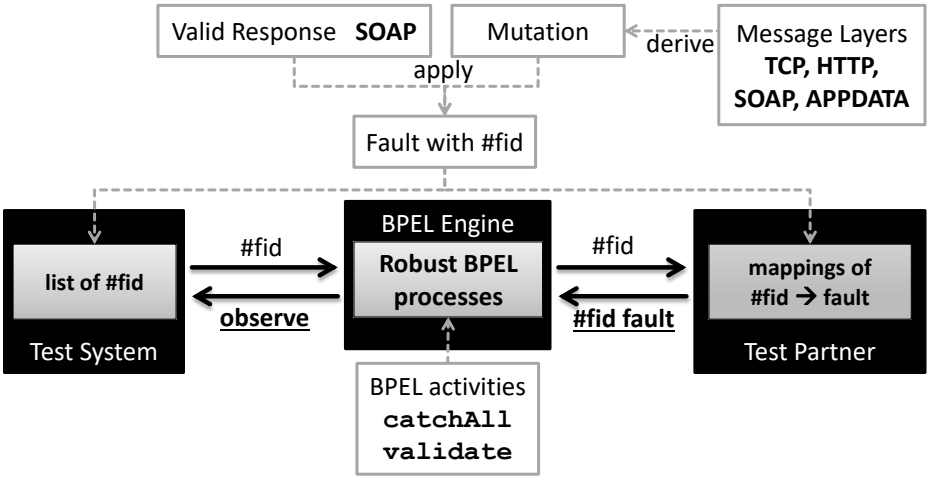


Figure 4.16.: BPEL Robustness: Big Picture

Big Picture

The big picture of the benchmark creation and its resulting testbed is shown in Figure 4.16. It comprises three components (black boxes) and their configuration (gray boxes): the test system, the BPEL engine under test, and the test partner. The test system requires the list of all fault IDs. Onto the BPEL engine, the robust processes need to be deployed, and the test partner has to be parameterized with the request-response mappings for pairs of the fault ID and the corresponding fault. The gray dotted arrows denote how the robust processes and how the faults and their fault IDs (#fids) are created. The robust processes are created by using activities from the process language that enable fault handling and error recovery. In contrast, the creation of the faults requires multiple steps. First, for each specification of each message layer, mutations are derived on how to change a valid message to an invalid one. Second, these mutations are applied to a normative valid response. Third, each fault is assigned an ID, a fault id, that corresponds to the fault, making up a request-response pair. The black arrows mark the message flow per test execution between the components. A test is initiated by the test system by sending a fault ID to a deployed robust process and observes the response which is used to determine the message robustness. The test partner, however, responds to requests of the process instances by returning the fault corresponding to the sent fault id.

Robust Processes

Two robustly designed processes are used, one being an extension of the other. They have been created using the process stub from the BPEL conformance benchmark in Section 4.5.2.1 using Stub Extension (P3). Their abbreviated XML serialization is shown in Listing 4.6, with variables marked via the dollar prefix and the actual behavior within the comments. Line 6 and 19 represents the *receive-reply* pair to observe the behavior of the *scope* encompassing the lines 7 to 18 in which the faulty partner service is called synchronously using the *invoke* activity and then sets the *$result* variable to *NO_FAULT*. To the *scope* itself, a *catchAll* fault handler is attached which sets the *$result* variable to *FAULT* in case it catches any fault. The *catchAll* activity is specified "to catch any fault not caught by a more specific fault handler" [181, p. 128], hence, it is the central point of creating a robust and fault tolerant process as it allows reacting upon an error. By checking the contents of the *$result* variable after process completion, we can observe whether the *catchAll* activity is executed at runtime or not, and consequently determine whether the engine was able to react upon the simulated error or not. The second robust process (RP#2) extends the first robust process (RP#1) by validating the received response against its XSD definition with the *validate* activity in line 15. This is an additional robustness instrument to ensure that the incoming message is validated against its XSD definition, independent of the engine which may or may not validate the incoming message against its XSD definition. The activities of RP#1 are widely used in real world processes. According to Hertis and Juric [106], "more than 70% of [real world] processes contain fault handlers [and the activities] invoke, sequence, assign and receive occur in more than 93% of processes" [106, p. 7]. The *validate* activity of RP#2 is not used in any of the real world processes stated by Hertis and Juric [106]. Hence, this indicates that that countermeasure is not widely applied. Although there is the *catch* activity as well to handle a single and specified fault, we cannot apply this activity as in our experiment we want to evaluate the ability to catch undefined and unspecified faults (i.e., no predefined SOAP faults). The processes are considered robust as they try to use fault handling logic to cope with an erroneous response by themselves without the intervention of an administrator.

Listing 4.6: Robust BPEL Process #1 and #2 in Pseudo XML.

```
1  <process>
2      <imports ... />
3      <partnerLinks ... />
4      <variables><!-- $id, $result, $response --></variables>
5      <sequence>
6          <receive/> <!-- set $id -->
7          <scope>
8              <faultHandlers>
9                  <catchAll>
10                     <assign /> <!-- set FAULT to $result -->
11                 </catchAll>
12             </faultHandlers>
13             <sequence>
14                 <invoke /><!-- sends $id and waits for (faulty) $response -->
15                 [only #2: <validate /><!-- validate $response against XSD -->]
16                 <assign /><!-- set NO_FAULT to $result -->
17             </sequence>
```

```
18          </scope>
19          <reply/> <!-- return $result -->
20      </sequence>
21  </process>
```

Decomposition

As already roughly outlined, the capability is decomposed into robustness approaches, message layers, and mutations. The approaches, message layers and the mutation count is shown in Table 4.8. The two robustness approaches based on the two robust processes are backdoor robustness and improved backdoor robustness. For the four different message layers TCP, HTTP, SOAP, and Application-Specific Data (APPDATA), mutations are specified. Table 4.8 shows only the mutation count and the actual mutations are listed in Table 4.9.

Table 4.8.: BPEL Robustness: Robustness Approaches, Message Layers, and their Mutation Count

Backdoor Robustness	75	**Improved Backdoor Robustness**	75
APPDATA	11	APPDATA	11
SOAP	21	SOAP	21
HTTP	40	HTTP	40
TCP	3	TCP	3

Mutations

To verify whether a BPEL engine under test has a satisfying degree of message robustness, 75 mutations are specified in Table 4.9. The mutations are structured according to the message layers in which an error may occur, being either on the lowest level (3 faults), in the HTTP header (40 faults), in the XML-based SOAP envelope (21 faults) or in the application specific part of the SOAP body that contains the APPDATA (11 faults).

TCP Mutations

The three mutations on the TCP layer refer to not being able to resolve the DNS entry and having either an unreachable or unresponsive host. These mutations are based on the faults defined by Kopp et al. [136]. They are simulated by changing the endpoint of the external partner service according to the specific fault, and are therefore the only three mutations that are not on the message level but the transport level.

HTTP Mutations

For the HTTP layer, we solely came up with mutations for different status codes of the HTTP header by sending the correct SOAP payload but only changing the status code. As the first digit of the HTTP status code determines its type, we subdivided them accordingly. The selected status codes are taken from the HTTP 1.1 RFC 2616 [65]. Only the status code 200 is removed as this marks a correct response, which is not the intent to test as part of this benchmark. Despite the return code 306 is specified as unused by Fielding et al. [65], it is still included as it should not be accepted according to this specification. Although there are many other HTTP header properties (e.g., the MIME type [71, 72]), this work solely focuses on the status code. Other properties are left for future work.

Table 4.9.: BPEL Robustness: Message Layers, and their Mutations

Layer	Mutation	Layer	Mutation
APPDATA	remove XML element	SOAP	no XML root element
	remove content		text instead of XML root
	change int to string		two XML root elements
	change int to double		elements overlap
	change int to loc. double		unclosed XML attribute
	remove XML namespace		unclosed XML element
	wrong XML namespace		unclosed XML comment
	unbound XML ns prefix		unclosed CDATA
	add XML element		unescaped lesser than
	add XML attribute		unescaped greater than
	add text between XML elements		unescaped ampersand
			unescaped apostrophy
HTTP	status code 100 - 101		unescaped quotation
	status code 201 - 206		XML name starts with XML
	status code 300 - 307		XML name starts with number
	status code 400 - 417		XML name starts with dash
	status code 500 - 505		XML name contains space
			remove XML element
TCP	DNS unresolvable		remove XML namespace
	host unreachable		wrong XML namespace
	response timeout		unbound XML ns prefix

SOAP & APPDATA Mutations

The mutations for both, the SOAP and the APPDATA layer, are mutations of a valid SOAP response. The XML mutations, which refer to the well-formedness criteria of XML, are solely at the SOAP layer while we subdivided the XSD mutations, which refer to the correctness against their XSD schema, into the part referring to SOAP XSD schema and the XSD schema of the application specific code. The XML mutations are extracted out of the XML specification [273, section 2.1] and grouped the mutations by bad names (4 mutations), unescaped symbols (5 mutations), unclosed entities (4 mutations), structural errors (1 mutations) and root element issues (3 mutations). For the XSD mutations, the operator action (i.e., add, remove, and change) and the operator target (namespace, namespace prefix, element, content, attribute, and text) have been permutated, meaningless combinations have been removed and mutations have been created for the meaningful ones. The issue is, however, that not every meaningful combination can be applied to both the SOAP and the APPDATA variant as not every mutation makes sense. Therefore, the SOAP XSD mutations are fewer than the ones for the APPDATA layer.

Example

For instance, the mutation *no XML root element* is part of the SOAP message layer and used in both robustness approaches: backdoor robustness and improved backdoor robustness.

Limitations

A limitation of this benchmark is that the robustly designed BPEL processes only make use of the forward error recovery activities through *catchAll*. The

benchmark could be extended to handle backward error recovery activities as well, especially in the case when forward error recovery fails to work.

The relevant metrics count how many mutations there are and how many of them are successful, timed out, or ignored the fault. Based on these metrics, the fault handling strategy can be determined per message layer and overall. What is more, attached to the language, the different robustness approaches can be compared and it can be checked whether the improved backdoor robustness is better than the normal backdoor robustness. Metrics

Feature to Test Method for Robustness Next, the tests are created for each mutation. An example is given in Listing 4.7. The test case (line 3–19) is the same for any test, only the fault ID that is sent to the deployed process instance changes (line 15). The test partner, however, differs from test to test as it defines how the fault is injected. For that, a rule-based test partner is used that checks for the number in the received message and reacts according to the mutation (line 21–34). The predefined responses are manually created by applying the mutations to a valid response. For the valid response, a special test is made acting as a baseline test to determine that the setup is working. In the example, the response is an empty one as the mutation states that there is no XML root element (line 29). F2TM

Listing 4.7: BPEL Robustness: Test for Mutation *no XML root element* of the SOAP Message Layer for the Backdoor Robustness Approach

```
1  <test feature="BR_ERR60001" id="BR_ERR60001__test">
2      <process>BR_ERR60001_soap-xml-root_elem-none.bpel</process>
3      <testCases>
4          <testCase name="Good-Case-1" number="1">
5              <testSteps>
6                  <checkDeployment>
7                      <testAssertions><assertDeployed/></testAssertions>
8                  </checkDeployment>
9                  <sendSoapMessage>
10                     <testAssertions>
11                         <assertXpath expression="..." expectedOutput="-1"/>
12                     </testAssertions>
13                     <operation name="startProcessSync" isOneWay="false"/>
14                     <service name="testInterface"/>
15                     <message><!-- send 60001 --></message>
16                 </sendSoapMessage>
17             </testSteps>
18         </testCase>
19     </testCases>
20     <files>TestInterface.wsdl TestPartner.wsdl</files>
21     <testPartners>
22         <ruleBasedWSDLTestPartner>
23             <wsdl>TestPartner.wsdl</wsdl>
24             <rules>
25                 <rule operation="startProcessSync">
26                     <xpathPredicate><!-- is 60001 ? --></xpathPredicate>
27                     <soapMessageOutput statusCode="500"
28                                        mimetype="application/soap+xml">
29                         <!-- empty -->
30                     </soapMessageOutput>
31                 </rule>
32             </rules>
33         </ruleBasedWSDLTestPartner>
34     </testPartners>
35 </test>
```

Threats to Validity

The threats to validity of creating this benchmark concern the manual creation of the tests. Especially the implementation of the SOAP and APPDATA tests required manual modeling of errors. To reduce as many flaws in these tests as possible, we applied Open Sourcing (P5), Expert Review (P6), and Automatic Static Analysis (P7). Automatic Static Analysis (P7) is performed by checking that only the introduced fault is found in the fault-injected message and no other fault by accident. For instance, the XMLlint[53] tool is used to determine that the injected fault in the SOAP message is correctly detected by the state-of-the-art XML validation tool.

Metrics

The standard metrics of the test are not sufficient for the robustness benchmark. Although the *testSuccessful* result is helpful in determining whether the process engine allows the process instance to react upon an error, the reason is unclear if it is not successful. It, however, is important to know whether there has been a timeout or whether the fault is simply being ignored by the process instance. Hence, the additional metric *testRobustnessResult* is introduced which is either $+$ in the case of a successful test, T in case the test system observes a time out, and R if the fault is ignored and the regular response is sent to the test system.

4.5.3. BPMN-based Benchmarks

The BPMN-based benchmarks cover the capabilities conformance, expressiveness, static analysis, and performance. After presenting the general approach to observing instances of BPMN process models, each benchmark is detailed separately.

Observation

Although BPMN supports the exchange of well-defined messages, actual and/or standard-based support for that in BPMN engines is not available. Because of this, in contrast to BPEL, the BPMN-based benchmarks use execution traces to observe and assert behavior instead of messages. This is called Execution Trace Evaluation (P14) and comprises *actions* and *traces*. Three different kinds of actions and traces exist. Actions can create log traces directly, indirectly, or none at all. Similarly, traces can be caused by actions directly, indirectly, or not at all. An overview of the actions and traces is given in Table 4.10. They are used in the upcoming benchmarks. The first set of traces and actions is the one in which an action directly causes a trace. In the second set, the actions cannot be asserted through traces at all. The third set comprises traces that are computed by the *gather traces* test step by analyzing the logs of the engine, but not caused by any action directly. The last and fourth set contains the actions that cause side effects which are later converted to traces through the *gather traces* test step if applicable, and then evaluated. Each action is executed as part of a Java or Groovy script within a BPMN script task. In the following, all actions and traces are detailed.

[53] `http://xmlsoft.org/xmllint.html`, visited 2017-3-31

Table 4.10.: Trace Assertions and Actions used in the Benchmark

action with direct trace	traceless action	gathered trace
SCRIPT_task1	WAIT_TEN_SECONDS	ERROR_deployment
SCRIPT_task2	CREATE_LOG_FILE	ERROR_runtime
SCRIPT_task3		ERROR_processAborted
SCRIPT_task4		ERROR
SCRIPT_task5		

action(s)	gathered trace
CREATE_MARKER_FILE	MARKER_exists
THROW_ERROR	ERROR_thrownErrorEvent
CREATE_TIMESTAMP_LOG_1	EXECUTION_parallel
CREATE_TIMESTAMP_LOG_2	
SET_STRING_DATA	DATA_correct
LOG_DATA	
INCREMENT_INTEGER_VARIABLE	INCREMENT_correct
INCREMENT_INTEGER_VARIABLE_AND_LOG	

Most tests rely on the action *CREATE_LOG_FILE* to create a log at the beginning of the process model and make use of one or more of the five *SCRIPT_taskX* actions that write their counterpart into that previously created log. In some situations, these actions and traces are not enough. The other actions and traces that provide more specific observation functionality are described in the following. Create Log & Write Tasks

The action *WAIT_TEN_SECONDS* simply waits ten seconds. This is necessary for three tests with *SubProcesses* to synchronize that when a *SubProcess* emits an event the listener in the parent *Process* is already active. Wait

Errors are expressed through one of the following three traces: *ERROR_deployment*, *ERROR_runtime*, and *ERROR_processAborted*. Each of them is automatically computed during the gather traces test step by analyzing the logs and querying the engine API. Hence, both Detailed Logs (P17) and Engine API Evaluation (P15) is applied here. If only the general presence of an error has to be asserted, the *ERROR* trace can be used. Error

Seven tests have two processes within the BPMN file: a main process and a secondary one. The main process uses the typical actions and is the target of the test which should be evaluated. Hence, the traces correspond to this process alone. Regarding the secondary process, it is only relevant whether it has been executed or not. To check this, the secondary process can *CREATE_MARKER_FILE* which is checked during the *gather traces* test step and, if available, would result in the *MARKER_exists* trace. Marker

An error is hard to reproduce with native standard compliant BPMN elements. It, however, can be thrown using THROW_ERROR and handled with native BPMN elements within a process. The *gather traces* step determines Error

whether any uncaught or unhandled error event has occurred and adds the trace *ERROR_thrownErrorEvent* to the log if it does.

Concurrency Detection

Based on the Execution Trace Evaluation (P14), it is also possible to implement Concurrency Detection (P16). In that case, each parallel activity creates a pair of timestamped log traces through either the *CREATE_TIMESTAMP_LOG_1* or the *CREATE_TIMESTAMP_LOG_2* action. In between the creation of the timestamp pair, the action simply waits ten seconds to ensure that there can be observable concurrency. The first trace is written upon entering and the second upon exiting the parallel activity in a separate file solely for these timestamps. The two actions must be used in different parallel branches, e.g., one for each flow of a parallel gateway. If timestamped log trace pairs overlap, concurrency is detected, and the *EXECUTION_parallel* trace is created in the dedicated log.

Data

With the actions *SET_STRING_DATA* and *LOG_DATA*, it is possible to store a specific value within a string property or log its current value to a separate file. In the gather traces step, this separate file is evaluated and, in case the file contained a specific value, the *DATA_correct* is written to the execution trace. This works similarly for an integer variable as well through *INCREMENT_INTEGER_VARIABLE* and *INCREMENT_INTEGER_VARIABLE_AND_LOG* which would cause an *INCREMENT_correct* in the case of success. The only difference is that the second action increments the variable and logs it. This is necessary to test some language elements such as script tasks with loop characteristics.

Engine-Dependent Actions

The actions are engine-dependent and have to be transformed to engine-specific implementations. This is not detailed here, but part of the Process Engine Benchmark Framework (PEBWORK) prototype described in Section 5.4.

4.5.3.1. BPMN Conformance Benchmark

Parts of this section have been taken from [81, 84].

The first benchmark of BPMN evaluates the capability conformance. Similarly to the BPEL evaluation, it measures part of the functional suitability quality characteristic and its two sub-characteristics functional completeness and functional correctness.

C2FM

Capability to Feature Method for Conformance The capability is hierarchically decomposed by relying on the existing structure of the BPMN specification [115]. The constructs of BPMN are the feature sets which are organized in construct groups, and each construct configuration represents a testable feature. An overview focusing on the constructs is given in Table 4.11. There are five construct groups, 33 constructs, and 105 construct configurations.

Construct Groups

The five construct groups are called *gateways* [115, Chapter 10.6], *events* [115, Chapter 10.5], *activities* [115, Chapter 10.3], *data* [115, Chapter 10.4], and *basics* [115, Chapter 10.2]. The BPMN element that is part in every process

model is the *SequenceFlow* which connects any "nodes" in the control flow graph of a process. This connector comes in various forms, and can carry a condition as well. Also, this group contains the constructs *Lane* and *Participant*. In the BPMN specification, there are three different events: *start*, *intermediate*, and *end* events. An event can reference zero, one, or more event definitions, such as, *cancel*, *compensation*, *conditional*, *error*, *escalation*, *link*, *message*, *signal*, *terminate*, and *timer* event definitions. Not every event-definition combination, however, is allowed [115, p. 259–260]. To fight test explosion, multiple event definitions are covered exemplary only. The activities group comprises *Activities* (e.g., *SendTask*, *ReceiveTask*, *LoopTask*, or *MultiInstanceTask*), *SubProcesses*, and *CallActivities*. The last two belong together because a *CallActivity* must invoke a particular *(AdHoc)SubProcess*. Gateways influence the control flow in a BPMN process and are either *exclusive*, *inclusive*, *parallel*, *event-based*, or *complex* gateways. Combinations of them are covered as well. In the data group, the BPMN constructs can define variables and offer read and write facilities. In this case, only the two simple constructs available from the specification are added to the benchmark: *DataObject* and *Property*. They are used to declare data storages within a process instance, which can be read and written, for instance, through script tasks using engine-specific scripts.

Table 4.11.: BPMN Conformance: Groups, Constructs, and Number of Construct Configurations

gateways	14	**events**	56	**activities**	27
ExclusiveGateway	3	Cancel Event	1	CallActivity	2
MixedGatewayCombi.	4	Compensation Event	6	MultiInstanceTask	8
InclusiveGateway	2	Conditional Event	5	LoopTask	6
ParallelGateway	2	Error Event	4	SubProcess	1
ComplexGateway	1	Escalation Event	7	Transaction	1
EventBasedGateway	2	Link Event	1	AdHocSubProcess	2
		Message Event	3	TokenCardinality	4
		Signal Event	9	SendTask	1
		Terminate Event	1	ReceiveTask	2
		Timer Event	9		
		Multiple Events	6		
		EventDefinitionRef	4		
data	2	**basics**	6		
DataObject	1	Lanes	1		
Property	1	Participant	1		
		SequenceFlow	1		
		SequenceFlow Cond.	3		

Example

For example, the exclusive gateway construct of BPMN is a construct which is part of the gateways group and has two configurations, namely, 1) the exclusive gateway with sequence flows having condition expressions (*Exclus-*

iveGateway) and 2) the exclusive gateway with one sequence flow marked as default (*ExclusiveGateway-Default*).

Metrics The metrics are the same as the one used by the BPEL conformance benchmark in Section 4.5.2.1 for both the feature tree and the tests. Hence, it is not detailed again.

Limitations A conformance benchmark for process engines can only contain the elements which can be executed (i.e., which are covered by the BPMN execution semantics [115, Chapter 13]). This benchmark comprises a majority of the available language constructs of BPMN, but not all of them. Consequently, it is not considered complete, lacking evaluation of advanced data handling constructs and the sending/receiving of SOAP messages via WSDL-based WSs. These two aspects were omitted because the engines that are part of PEAL do not support these in a standard-conformant way. This is problematic because it means that At Least One Success (P21) cannot be applied to verify the correctness of a test.

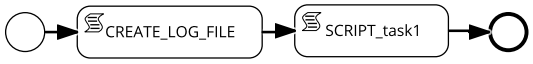


Figure 4.17.: BPMN Conformance: Process Model Stub

F2TM **Feature to Test Method for Conformance** To create the tests for the construct configurations (i.e., features), Stub Extension (P3) is applied using an existing BPMN process model shown in Figure 4.17. It can be seen that the actions *CREATE_LOG_FILE* and *SCRIPT_task1* have been used. The corresponding test would need the *gather traces* test step that uses the *SCRIPT_task1* assertion, complementary to the *SCRIPT_task1* action.

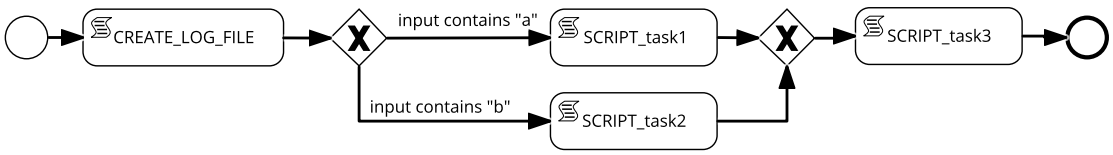


Figure 4.18.: BPMN Conformance: Process Model for the *ExclusiveGateway* Test

Example: Exclusive Gateway The test of the *ExclusiveGateway* construct configuration comprises the process model depicted in Figure 4.18 and the test serialization shown in Listing 4.8. It is a process with four *scriptTasks* and an *exclusiveGateways*. The execution of two of the tasks is controlled by the *exclusiveGateways* and only one of the tasks is executed. To cover the combinations, four different test cases have been created injecting either *a* (line 4–22), *b* (line 23), *ab* (line 24), or *c* (line 25) with different expected traces depending on the injected input.

Listing 4.8: BPMN Conformance: PEBL Test for *ExclusiveGateway*

```
<test feature="Conformance__ExclusiveGateway" id="
    Conformance__ExclusiveGateway__test">
    <process>ExclusiveGateway.bpmn</process>
    <testCases>
        <testCase number="1">
            <testSteps>
                <checkDeployment>
                    <testAssertions><assertDeployed/></testAssertions>
                </checkDeployment>
                <startProcess>
                    <variables>
                        <variable name="test" type="String" value="a"/>
                    </variables>
                    <processName>ExclusiveGateway</processName>
                </startProcess>
                <gatherTraces>
                    <testAssertions>
                        <assertTrace trace="SCRIPT_task1"/>
                        <assertTrace trace="SCRIPT_task3"/>
                    </testAssertions>
                </gatherTraces>
            </testSteps>
        </testCase>
        <testCase number="2"> <!-- "b"  -> "SCRIPT_task2,SCRIPT_task3" --></...
            >
        <testCase number="3"> <!-- "ab" -> "SCRIPT_task1,SCRIPT_task3" --></...
            >
        <testCase number="4"> <!-- "c"  -> "ERROR_runtime" --></...>
    </testCases>
    <testPartners/>
</test>
```

4.5.3.2. BPMN Expressiveness Benchmark

Parts of this section have been taken from [84].

The second benchmark for evaluating BPMN engines targets the capability expressiveness which is part of the quality characteristic functional suitability and its sub-characteristics functional completeness and functional correctness.

Capability to Feature Method for Expressiveness The patterns are the feature sets which are organized in pattern catalogs, and each pattern implementation represents a testable feature. Similarly to the expressiveness of BPEL, the original 20 workflow control-flow patterns from van der Aalst et al. [257] are used, and no extensions or derivations thereof, since these are most widely known. We built upon the pattern-based analysis for BPMN 1.0 presented by Wohed et al. [287]. Most of the pattern implementations described in the paper can directly be applied to BPMN. In the rare cases, where a modification of a pattern implementation was necessary, the rationale of Wohed et al. [287] was followed to provide a solution. C2FM

Table 4.12 lists the patterns [257, 287] sorted by the pattern number, along with the highest degree of pattern support that can be achieved for BPMN. The degree of support that is possible in BPMN is based on Wohed et al. [287]. It can be seen in Table 4.12 that two patterns (MI without A Priori Run-Time Knowledge and Milestone) cannot be directly implemented in BPMN Patterns

as no workarounds based on the extended vocabulary of BPMN that could compensate for this have been found. Hence, these patterns are excluded from further discussion. For the remaining patterns, there is at least a single implementation, according to the structures from Wohed et al. [287], which led to at least partial pattern support.

Example For instance, the pattern *WCP04 exclusive choice* is part of the workflow control flow pattern catalog group and has a single pattern implementation. The pattern *WCP06 multi choice* has three pattern implementations, namely, one using the inclusive gateway, one the complex gateway, and one the implicit definition of the control flow through multiple outgoing sequence flows.

Table 4.12.: BPMN Expressiveness: Pattern Catalog, Patterns, and Number of Pattern Implementations

Control-Flow Pattern Catalog[257]		**BPMN**	23
WCP-01	Sequence	+	1
WCP-02	Parallel Split	+	1
WCP-03	Synchronization	+	1
WCP-04	Exclusive Choice	+	1
WCP-05	Simple Merge	+	1
WCP-06	Multi-Choice	+	3
WCP-07	Structured Synchronizing Merge	+/-	1
WCP-08	Multi Merge	+	1
WCP-09	Structured Discriminator	+/-	2
WCP-10	Arbitrary Cycles	+	1
WCP-11	Implicit Termination	+	1
WCP-12	MI Without Synchronization	+	1
WCP-13	MI With A Priori Design-Time Knowledge	+	1
WCP-14	MI With A Priori Run-Time Knowledge	+	1
WCP-15	MI Without A Priori Run-Time Knowledge	-	0
WCP-16	Deferred Choice	+	1
WCP-17	Interleaved Parallel Routing	+/-	1
WCP-18	Milestone	-	0
WCP-19	Cancel Activity	+/-	1
WCP-20	Cancel Case	+	3

F2TM **Feature to Test Method for Expressiveness** The tests are created using Execution Trace Evaluation (P14), analogous to the conformance tests for BPMN. The standard metrics are sufficient. As the tests are so similar to the BPMN conformance ones in Section 4.5.3.1, they are not detailed again.

4.5.3.3. BPMN Static Analysis Benchmark

This benchmark has originally been created by Geiger et al. [85]. The author of this work helped in creating a PEBL serialization for that benchmark.

The third BPMN benchmark evaluates the capability static analysis. It makes the quality sub-characteristic user error protection, which is part of the quality characteristic usability, measurable.

Capability to Feature Method for Static Analysis BPMN itself does not have explicitly enumerated static analysis rules as part of the language specification. However, the rules are implicitly defined within the specification as part of the prose. Geiger and Wirtz [79] created models that explicitly violate these rules as well as a tool to detect these violations called BPMNspector [86]. Based on these erroneous models, a benchmark has been created [85]. The static analysis rules are the feature sets which are organized in rule sets, and each rule configuration represents a testable feature. An overview of the rules and the number of different rule configurations are found in Table 4.13. As shown, there are 151 rules and 301 rule configurations in total. C2FM

For instance, the rule *EXT002* is part of the BPMN constraints group and contains a single rule configuration with the violation of containing two elements with the same ID (i.e., duplicate ID). Example

Feature to Test Method for Static Analysis The static analysis tests also make use of Execution Trace Evaluation (P14) but they only rely on the trace computation based on the engine logs and the engine API to detect whether the deployment has failed (*ERROR_deployment*). An example of such a test is given in Listing 4.9. In the sole test case, the traces are gathered immediately to determine whether the deployment has failed. Each test serialization is the same for each rule configuration, except for the referenced process model and the test ID. F2TM

Listing 4.9: BPMN Static Analysis: Example Test using PEBL

```
<test feature="EXT127_3_failure_intCatch" id="EXT127_3_failure_intCatch__test">
    <process>EXT127_3_failure_intCatch.bpmn</process>
    <testCases>
        <testCase name="Good-Case-1" number="1">
            <testSteps>
                <gatherTraces>
                    <testAssertions>
                        <assertTrace trace="ERROR_deployment"/>
                    </testAssertions>
                </gatherTraces>
            </testSteps>
        </testCase>
    </testCases>
    <testPartners/>
    <extensions>
        <staticAnalysisChecks>false</staticAnalysisChecks>
    </extensions>
</test>
```

Table 4.13.: BPMN Static Analysis: Rules and Rule Configurations

rule	301	**rule**		**rule**		**rule**		**rule**		**rule**	
EXT002	2	EXT029	1	EXT054	1	EXT082	1	EXT106	2	EXT130	1
EXT003	1	EXT030	1	EXT055	1	EXT083	1	EXT107	1	EXT131	1
EXT004	1	EXT031	4	EXT056	4	EXT084	1	EXT108	1	EXT132	1
EXT006	3	EXT032	1	EXT057	2	EXT085	1	EXT109	1	EXT133	1
EXT007	3	EXT033	1	EXT058	2	EXT086	1	EXT110	8	EXT134	1
EXT008	3	EXT035	1	EXT059	6	EXT087	3	EXT111	2	EXT135	3
EXT009	3	EXT036	3	EXT060	1	EXT088	1	EXT112	1	EXT136	1
EXT010	1	EXT037	1	EXT061	1	EXT089	1	EXT113	1	EXT137	1
EXT011	3	EXT038	1	EXT062	3	EXT090	1	EXT114	1	EXT138	3
EXT012	10	EXT039	1	EXT063	5	EXT091	1	EXT115	2	EXT139	1
EXT013	4	EXT040	1	EXT064	2	EXT092	1	EXT116	2	EXT140	1
EXT014	1	EXT041	1	EXT065	2	EXT093	5	EXT117	2	EXT141	1
EXT017	1	EXT042	1	EXT067	1	EXT094	5	EXT118	1	EXT142	1
EXT018	2	EXT043	1	EXT068	1	EXT095	2	EXT119	4	EXT143	1
EXT019	2	EXT044	2	EXT069	2	EXT096	1	EXT120	1	EXT144	1
EXT020	4	EXT045	2	EXT070	2	EXT097	2	EXT121	1	EXT145	1
EXT021	1	EXT046	1	EXT071	1	EXT098	8	EXT122	1	EXT146	5
EXT022	1	EXT047	1	EXT072	1	EXT099	2	EXT123	2	EXT147	8
EXT023	2	EXT048	2	EXT076	3	EXT100	3	EXT124	1	EXT148	6
EXT024	1	EXT049	1	EXT077	1	EXT101	1	EXT125	1	EXT149	5
EXT025	2	EXT050	1	EXT078	1	EXT102	1	EXT126	1	EXT150	4
EXT026	2	EXT051	1	EXT079	1	EXT103	1	EXT127	5	EXT151	4
EXT027	2	EXT052	2	EXT080	1	EXT104	1	EXT128	5	EXT152	2
EXT028	1	EXT053	1	EXT081	1	EXT105	2	EXT129	1	REF	17

Extensions & Metrics

The extension element is similar to Section 4.5.2.3 as it indicates that the standard static analysis checks should not be applied to evaluate the correctness of the process model. If a test would violate more than a single rule, it could be stated as an extension element as well. Regarding the metrics, and analogous to the BPEL static analysis benchmark, only the *deploymentSuccessful* metric is necessary.

4.5.3.4. BPMN Performance Benchmark

This benchmark has originally been created by Skouradaki et al. [237]. The author of this work helped in creating a PEBL serialization for that benchmark.

The last benchmark for BPMN covers the capability performance. This benchmark makes the quality characteristics performance efficiency with its sub-characteristics time behavior, resource utilization, and capacity defined in the ISO/IEC 25010 Quality Model [113] measurable. The benchmark and its results are already expressed in a DSL that comes with BenchFlow [61, 237]. This DSL deviates from PEBL because of its sole focus on performance. In the

following, the BenchFlow DSL is described followed by a mapping from the BenchFlow DSL to PEBL.

BenchFlow DSL

The BenchFlow DSL[54] is structured into *trials* and *experiments* [61, 237]. In a trial, one or more BPMN process models are deployed to the process *engine* under test and tested using a given *load function* (i.e., workload) while measuring *metrics* (e.g., CPU and RAM usage, throughput in the form of number of executed workflow instances per second and duration/number of the workflow instance). Trials which test the same process model using the same workload on the same environment are aggregated into experiments. Each experiment comprises three trial runs on three engines [237]. Because there are different workloads, there are multiple experiments. A more extensive version of their DSL has been created by Skouradaki et al. [236] as part of a DSS.

Differences & Mapping

The differences between PEBL and the BenchFlow DSL are threefold: 1) the definition of a feature, feature set, and group, 2) the description of additional domain-specific concepts such as load functions and environments, and 3) the observability of a test run. In BenchFlow, there are experiments and trials, whereas PEBL contains groups, feature sets, and features. The trials, however, do not contain helpful information for the interested users but are merely used so that the aggregated values measured in the experiment are stable and that any erratic and random outside influences on the measurements are minimized. There is, however, no grouping of the experiments available in BenchFlow. Therefore, the suggested mapping is to see the experiment as the feature and abstract away from the trials. What is more, BenchFlow has elements in its DSL for describing complex test environments comprising Docker containers and their interaction. As a result, information about the actual hardware, software setup, and configuration is described in the results of BenchFlow in much more detail through domain-specific elements. Such information cannot be captured through domain-specific elements directly in PEBL. Nevertheless, it can be stored in PEBL using the extension element instead. Although instances of the process models are started similarly by passing in a list of variables in both PEBL and BenchFlow, the latter does not use Execution Trace Evaluation (P14) but relies on Engine API Evaluation (P15). PEBL, however, cannot express test cases with Engine API Evaluation (P15) directly. Nevertheless, it is possible to use the *start process* test step and rely on the *execute script* test step for the other ones instead.

C2FM & F2TM

Because experiments cannot be aggregated in a sensible way, the feature sets and groups are not useful in that case. As the feature tree structure has to be fulfilled, placeholders for groups and feature sets are used. For instance, the experiment *micro benchmark* is part of the *default* feature set which is part of the *default* group so that it fits the expected tree-like structure. Within this experiment, the metrics are measured as aggregates. This means that instead

[54]The Cassandra database schema is available at `https://github.com/benchflow/docker-images/blob/dev/cassandra/data/benchflow.cql`, visited 2017-3-31

of providing raw numbers, only the average, minimum, maximum, standard deviation, and relative standard deviation are stored. The load function is captured through extensions.

4.5.4. Benchmark Results

After having evaluated whether PEBL is expressive enough to represent *benchmarks*, in this section, it is evaluated whether PEBL can represent *benchmark results* as well. Benchmark results comprise engines, test results, and aggregated results. As their structure is similar for any of the previously defined benchmarks regardless of the capability and process languages, the results are shown exemplary for the workflow pattern *WCP04 exclusive choice* on the BPEL engine Apache ODE 1.3.6.

Engine

The representation of Apache ODE 1.3.6 is given in Listing 4.10. Apart from the name (line 2), version (line 3), supported process language (line 5), and configuration options (line 4), additional information (line 6–12) is given as well. It comprises the Software Package Data Exchange (SPDX) license IDs[55], release date, the programming language in which they are built in, and the URL where to find the software. This additional information reveals internals of the engine which can help end users when using and developers when extending the engine. It is encoded through the use of extensions, as it is just additional but not essential information for the results of a benchmark.

Listing 4.10: PEBL Serialization of Engine Apache ODE 1.3.6

```
1  <engine id="ode__1_3_6">
2     <name>ode</name>
3     <version>1.3.6</version>
4     <configuration/>
5     <language>BPEL__2.0</language>
6     <extensions>
7        <license>Apache-2.0</license>
8        <licenseURL>http://spdx.org/licenses/Apache-2.0.html</licenseURL>
9        <releaseDate>2013-10-12</releaseDate>
10       <programmingLanguage>Java</programmingLanguage>
11       <url>http://ode.apache.org/</url>
12    </extensions>
13 </engine>
```

Test Result

In Listing 4.11, the test result for the WCP04 exclusive choice test is shown. The extension element is not necessary for the test result, as the domain-specific elements do suffice to represent the test results. The only exception is for the performance capability, in which the extension element contains the environment in which the test is executed in. This is important in that case, as the test results of performance tests heavily depend on the environment. The test result representation in Listing 4.11 includes the engine-dependent files such as the deployment descriptor as well as the deployment package and the logs from both the engine as well as its servlet container Tomcat. We can see that the two test cases finished without any failure messages and that the

[55] `http://spdx.org/spdx-license-list/license-list-overview`, visited 2017-3-31

measurement of the *testSuccessful* metric shows that the test is successful in this specific example.

Listing 4.11: Abbreviated PEBL Serialization of Test Result of the pattern WCP04 on the BPEL engine Apache ODE 1.3.6 with betsy.

```
<testResult test="WCP04__pattern__impl__test"
            engine="ode__1_3_6" tool="betsy__1_0">
 <logFiles>
  logs\catalina.log
  logs\ode.log
 </logFiles>
 <deploymentPackage>pkg</deploymentPackage>
 <files>
  process\deploy.xml
  process\TestInterface.wsdl
  process\WCP04-ExclusiveChoice.bpel
  pkg\WCP04-ExclusiveChoice.zip
 </files>
 <measurements>
    <measurement metric="WCP04__pattern__impl__testSuccessful" value="true"/>
 </measurements>
 <extensions/>
 <testCaseResults>
    <testCaseResult name="Good-Case" number="1"></testCaseResult>
    <testCaseResult name="Good-Case" number="2"></testCaseResult>
 </testCaseResults>
</testResult>
```

Aggregated Result

The aggregated results are shown in Listing 4.12. The listing shows how the aggregated results for the *support* metrics are given for both the pattern implementation and the pattern itself. Both the pattern and its pattern implementation are directly supported (+).

Listing 4.12: Abbreviated PEBL Serialization of Aggregated Result of the pattern WCP04 on the BPEL engine Apache ODE 1.3.6 with betsy.

```
<aggregatedResult engine="ode__1_3_6" tool="betsy__1_0">
   <measurement metric="WCP04__pattern__support" value="+"/>
   <measurement metric="WCP04__pattern_implementation__support" value="+"/>
</aggregatedResult>
```

Summary

To sum up, PEBL can express the benchmark results through its domain-specific elements. The extension elements are helpful in adding more information, but they are only used sparsely for additional information that depends on the capability that is being benchmarked. Hence, it is not overused or abused, and therefore, the language can be seen as an extensible DSL that is flexible to cover benchmarks for engine capabilities and quality characteristics.

4.5.5. Good Benchmarks

Relevance

In this section, a short evaluation is already made on how this language can already help in creating good benchmarks. The benchmarks described above cover half of the ISO/IEC 25010 Product Quality Model [118]. This is visualized in Figure 4.19 with the covered quality characteristics and subcharacteristics marked in green and are styled bold. This reveals the relevance of the benchmarks as they can be rooted in the quality model which captures the quality requirements of software. Furthermore, such quality requirements are used for evaluating software, and in this work process engines.

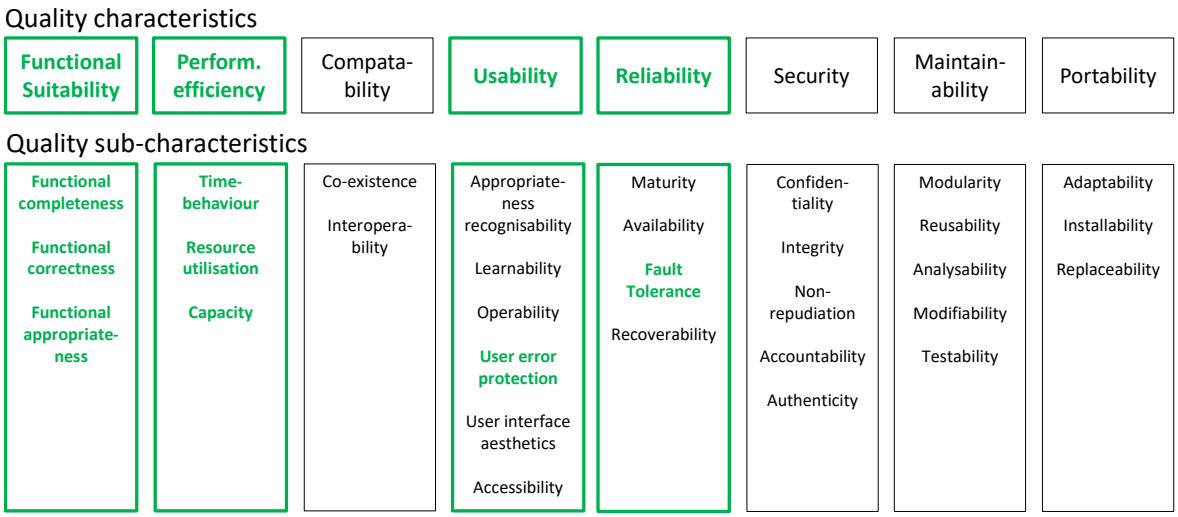


Figure 4.19.: Covered Quality Characteristics and Subcharacteristics from the ISO/IEC 25010 Product Quality Model [118]

Reproducibility, Clarity, and Portability

A good representation of a benchmark has to ensure that it captures the steps on how to conduct the benchmark (i.e., reproducibility), must be self-contained and precise (i.e., clarity), and must independent of the system under test (i.e., portability). Looking at PEBL, it can be seen that those three criteria are fulfilled. First, PEBL comprises domain-specific elements to describe the steps necessary to perform a benchmark and capture the results. Those steps can be executed over and over again, making the benchmark reproducible. Second, a serialization of PEBL comprises all information necessary to conduct a benchmark. Because of its domain-specific elements it is concise because most steps and information can be stated with a corresponding domain-specific element directly. Hence, a benchmark represented with PEBL is clear and concise. Third, the benchmarks themselves are described in a way so that they are engine independent, but language-dependent. Even language-independent benchmark concepts such as the workflow patterns require process language specific alterations and adaption. Consequently, the process engine independence seems to be a sufficient level of benchmark portability in this case. And PEBL can provide that level as well. So, in summary, PEBL enables reproducible, clear, and portable benchmarks (i.e., good benchmarks).

4.6. Summary

Summary

In this section, the Process Engine Benchmark Language (PEBL) is presented which can express benchmarks and their results. Its expressiveness is evaluated by crafting benchmarks for different process languages (i.e., BPEL and BPMN) and quality characteristics (i.e., functional suitability, performance efficiency, resilience, usability). These benchmarks themselves are relevant as they capture various relevant quality characteristics of process engines. The evaluation

showed that the expressiveness suffices to represent eight different benchmarks and their results in total. Even benchmarks from other approaches, e.g., BenchFlow [237], can be expressed as well. In other words, it is shown that the PEBL DSL can be used to describe approaches that measure metrics for a variety of quality criteria. Hence, hypothesis H4.2 *("A domain-specific testing language is a suitable form to make quality criteria measurable.")* is supported.

Future Work

Future work comprises three items. First, it is planned to overcome the limitations of the prototype. This includes adding the ability to process the Groovy-based aggregation scripts and a GUI to edit serializations of PEBL in a more user-friendly way. Second, a more extensive evaluation with more process languages (e.g., upcoming versions of BPEL or BPMN) as well as the (remaining) quality characteristics [113] (e.g., additional pattern catalogs in the expressiveness evaluation, or a complete conformance benchmark) is targeted. Although the current quality characteristics can already proof that PEBL is expressive for process engine benchmarks and their results, it is still somewhat limited, nevertheless. Third, PEBL shall be extended so that it includes more domain-specific elements that are available in other benchmarking DSLs in that domain (e.g., the DSL of BenchFlow [61]).

You don't have to be a genius or a visionary or even a college graduate to be successful. You just need a framework and a dream.

Michael Dell

5. Process Engine Benchmark Framework

Parts of this chapter have been taken from [81, 94, 95].

In this chapter, hypothesis H4.3 *("A benchmarking framework is a suitable means to reveal objective and well-founded information about process engines.")* is supported.

In this chapter, the Process Engine Benchmark Framework (PEBWORK) is outlined. It builds upon the Process Engine Benchmark Language (PEBL) and the Process Engine Abstraction Layer (PEAL) and produces results which can be visualized in the Process Engine Benchmarking Interactive Dashboard (PEBDASH). A more efficient version called Efficient Process Engine Benchmark Framework (ePEBWORK) is detailed in Chapter 7.

5.1. Motivation

Engine Selection Process

The engine selection process, as shown in Figure 5.1, is a collaboration of four different roles: the business analyst, the developer with domain knowledge of process engines, the benchmark framework, and the interactive dashboard with its loader. The process targets the selection of an engine as represented in both the start and the end event in Figure 5.1. First, the business analyst has to define the feature tree and the metrics he is interested in manually using PEBL. Second, the developer derives tests how to determine whether the features are supported, and provides scripts so that the metrics within the feature tree can be computed based on the metrics of the test results. Third, the benchmark framework executes the derived tests and produces test results. Fourth, the loader loads the produced test results along with the benchmark into the interactive dashboard and computes the metrics within the feature tree using the scripts written by the developer. Last, the business analyst then uses the computed metrics he previously had declared and that are visualized through the dashboard to make a decision which engine to choose. In the evaluation of PEBL in Section 4.5, approaches on how to decompose capabilities to features (C2FM) and make features testable (F2TM) are described, including the necessary metrics. This chapter is solely about the benchmark framework and its step to *execute tests*.

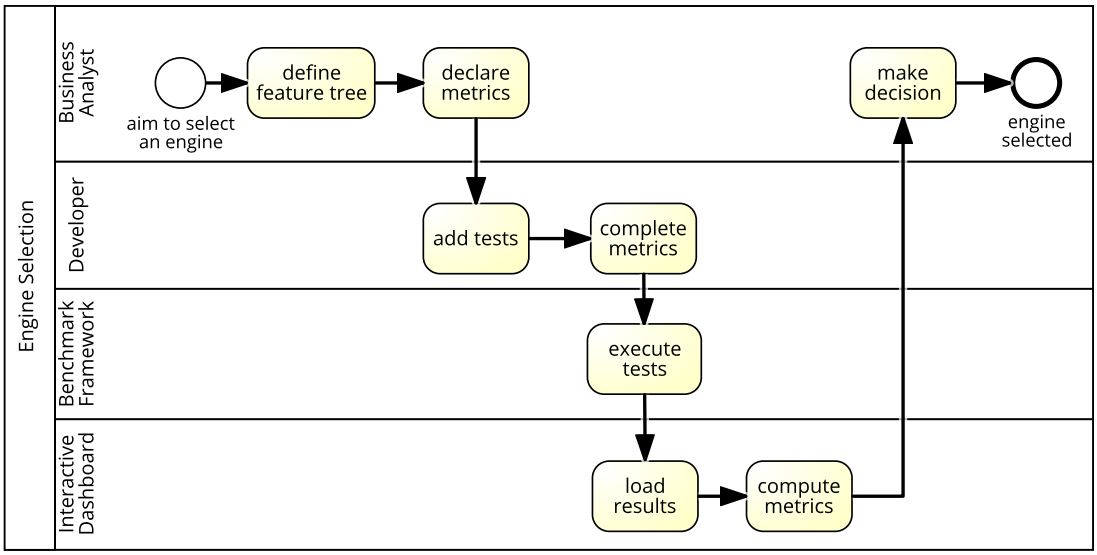


Figure 5.1.: Engine Selection Process

Requirements

The benchmark framework has to fulfill both, effectiveness and efficiency requirements. Its effectiveness is measured by its ability to produce valid results. Each test must produce its results reproducibly. This requires that everything has to be known upfront (R5.1). It is paramount for ensuring reproducibility that one test is executed in isolation to any other test (R5.2). Otherwise, a different result could be produced just by executing the tests in a different order. Last but not least, any intermediate artifacts must be provided to trace any fault in the test result to its origin (e.g., through logs) (R5.3). The efficiency is measured by the ability to complete a benchmark quickly. This requires first and foremost that the benchmark framework is fully automated (R5.4). Having such a fully automated framework also affects the practical effectiveness by excluding human error during the benchmark execution.

Chapter Structure

The remainder of the chapter is structured as follows. First, related work is presented in Section 5.2, followed by the framework design which is outlined through the data flow of the benchmarking procedure in Section 5.3. The prototype and its architecture that relies on a sequential form of the benchmarking procedure is described in Section 5.4, and the evaluation of the framework through experiments and theoretical criteria is detailed in Section 5.5. This chapter is concluded with a summary in Section 5.6.

5.2. Related Work

Related work comprises process engine benchmarking frameworks, frameworks that test processes running on process engines, and last, other related benchmarking or testing frameworks. Each of them is detailed in the following.

Engine Benchmarks & Frameworks

There are two process engine benchmarking frameworks available that are highly related to this work: Delgado et al. [45] and BenchFlow [61, 62]. Delgado et al. [45] provide a method to use test cases to reproducibly determine trivalent support results of process engine characteristics either through theoretical or practical means. Their execution, however, is not automated and no effort is made for test isolation. In contrast, BenchFlow [61, 62] is a much more complete and holistic competitor to this work as it also comprises a benchmark DSL, a uniform abstraction layer, and a fully automated framework to perform those benchmarks. It, however, only works for BPMN engines and it can only measure performance. Test isolation is guaranteed through the usage of Docker and creating fresh instances for each performance experiment. Although Rosinosky et al. [215] provide a framework for performance and cost evaluation of BPMS in the cloud, its primary focus is to benchmark cloud configurations of such BPMS. This work, however, is concerned with the quality characteristics of the process engines and not on the best cloud configuration for those process engines. Hence, their approach is not that relevant to this work. Another approach, that is not directly related is the one by Roller [213] who focuses on building a single, closed-source, and fast BPEL engine. To measure that the performance is as expected, he made a performance benchmark for his engine but he did not create a corresponding framework to execute benchmarks in general.

Process Testing Frameworks

In contrast to frameworks that focus on benchmarking or testing process engines, there are several frameworks that allow to test processes instead. BPELUnit [161, 168] is an xUnit-like unit testing framework for BPEL processes. And SOABench [17] as well as GENESIS2 [123] allow generating testbeds for testing SOA applications (e.g., BPEL processes). As part of the evaluation of SOABench and GENESIS2, BPEL processes on top of different BPEL engines have been evaluated. There are also BPMN-based testing frameworks available. An overview of the state of process model testing is given by Böhmer and Rinderle-Ma [23]. They have shown that most research is about test-case generation whereas process engine testing is not mentioned at all. Makki et al. [163] have created a test automation API for automated regression testing of BPMN process models. Their approach is based on a capture & replay paradigm with only a minimal overhead by instrumenting a BPMN engine.

Other Benchmarking & Testing Frameworks

Because process engines are important middlewares, benchmarking and testing frameworks of middlewares in general are related as well. ESBs are evaluated by Bhadoria et al. [15], including performance. However, the manual high-level evaluation measures performance as high, medium, or low [15, Table 6]. Testing middleware components (e.g., ESBs or Java Messaging Systems (JMSs)) under heavy load is an area of interest [140, 224, 245]. For ESBs, there is ESB Performance[56], a test suite created by AdroitLogic[57],

[56] `http://esbperformance.org`, visited 2017-3-31
[57] `http://www.adroitlogic.org/`, visited 2017-3-31

which has been around since 2007. In its seventh execution round, four ESBs were tested using SOAP/XML payloads via the HTTP protocol testing different message sizes and concurrent numbers of users. Regarding conformance testing, much work has been done in the area of Java Platform, Enterprise Edition (JEE). Oracle provides Test Compatibility Kits (TCKs) which can be used to evaluate whether a servlet container implements the servlet specification, however, provides no test isolation mechanisms.

5.3. Benchmarking Procedure

The benchmarking procedure contains and is represented only by the step *execute tests* as shown in Figure 5.1. This section focuses on the data flow: passing data items to functions.

Data Flow in BPMN

The data flow diagrams are modeled using the available elements and shapes for data flow modeling of BPMN [115] (see Section 2.2.2.2). Each data object in the data flow diagrams conforms to an element within PEBL. Moreover, each task must ensure that both their inputs and outputs conform to the specified schemas, and validate the integrity of them. Furthermore, IT systems change their internal state based on their input and produce output as well. No data stores are used in the diagrams. But, in any implementation, the data objects could be replaced with data stores to increase reuse.

Build System Metaphor

Any task may check automatically whether the proposed output has already been created for the given inputs and skip the execution of the task instead. This is similar to build tools like Gradle[58] or Apache Ant[59].

Levels

The step of the engine selection process is structured into two abstraction levels: the outsider view looking at the *execute tests* step from a high-level view and the insider view looking at the *execute tests* step from a detailed view, namely, from the inside. The relation between the two levels stems from simple hierarchic decomposition. In this case, a decomposition via sub-processes is applied which is supported natively in BPMN [115]. Hence, a more detailed description of this decomposition is neglected, but could easily be built based, for instance, on the process viewing patterns [228] or a process abstraction slider [199]. In the next sections, the different levels of the data flow are modeled and described.

5.3.1. Outsider View

IO System

From the outsider view, the task is a simple IO system (see Figure 5.2). It uses the tests defined in PEBL to compute the test results for engines specified through their engine IDs. Each test result contains a measurement of the

[58] `https://www.gradle.org`, visited 2017-3-31
[59] `https://ant.apache.org`, visited 2017-3-31

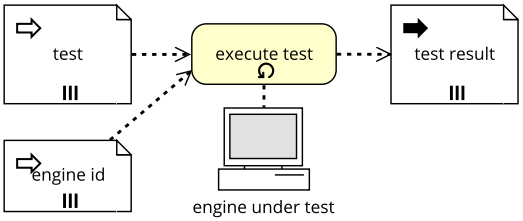


Figure 5.2.: The Data Flow of the Benchmark Framework of the Engine Selection Process - Outsider View

metrics defined in the tests. Measurements for the metrics defined by the business analysts (i.e., aggregated metrics) are not available yet, as they have to be computed based on these atomic measurements. The task *execute tests* can transform a single test on a single engine to a test result. By looping through all the combinations of the engine IDs and the tests, it produces $|engine\ IDs| \times |tests| = |test\ results|$.

5.3.2. Insider View

Three Phases

Switching from the outsider view in Figure 5.2 to the insider view, the task *execute tests* has been split up into three phases: setup, execute, and teardown. These phases are similar to typical testing frameworks, as they first prepare the test environment, then execute a single test, and finally, destroy the test environment. The execute phase connects both the setup and the teardown phase. In our case, the execute phase consists of only a single activity, whereas the other two phases are more complex. Hence, the process has been split up according to these two phases to better explain it, with the setup phase in Figure 5.3 and the teardown phase in Figure 5.4. The tasks are also color-coded into three categories. The yellow tasks do setup work, the gray task does actually *execute test cases* on the engine and the orange tasks do the teardown work. Apart from the inputs and outputs in these data flow models, there are also two types of IT systems: the test partners (colored gray) and the engine (colored blue).

Setup Phase

In the setup phase, the test bed to execute the given test on the engine with the given engine ID is provided. The engine-independent test has to be converted to an engine-dependent version for the given engine ID. This holds for both the test cases and the process model in the tasks *derive engine-dependent test cases* and *create deployable unit of process model*. An engine-dependent test case contains information how to interact with a specific engine to start the process and where to send the messages to. An engine-dependent and deployable process model refers to the deployment package defined in PEAL and contains all files in a package that can be directly deployed on a particular process engine. Most engines use their engine-specific format for such

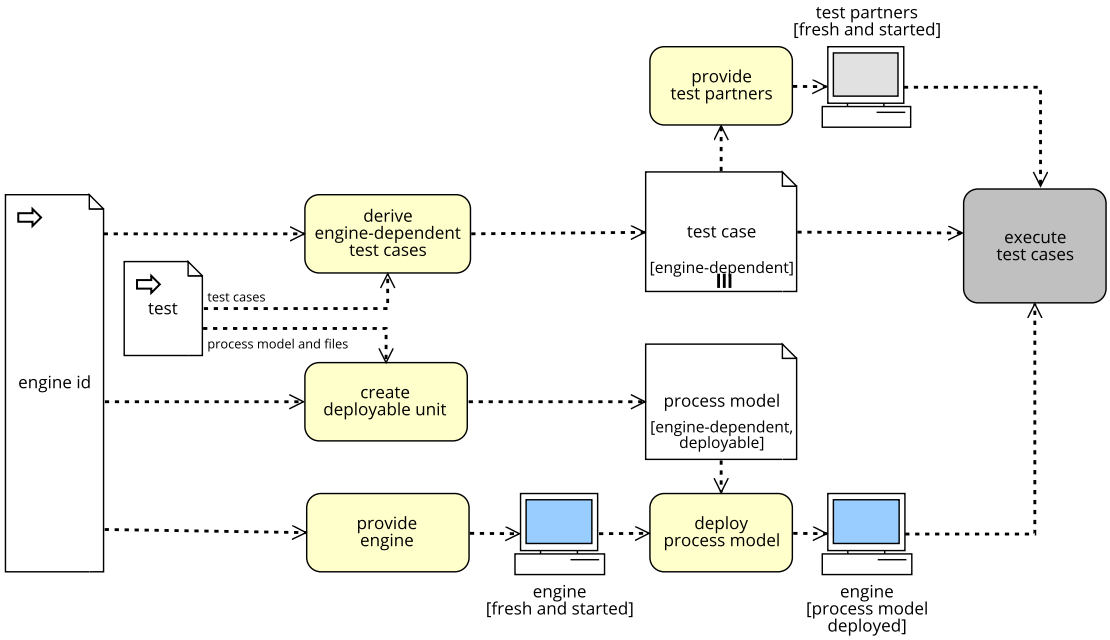


Figure 5.3.: Data Flow of the Benchmark Framework of the Engine Selection Process – Insider View of the Setup Phase

a deployment structure. Hence, they require engine-specific adaptation. An overview of these can be found as part of the prototype of PEAL in Section 3.4.2. Next, a fresh and started engine is provided (*provide engine* task) on which the previously created engine-dependent and deployable process model is deployed to (*deploy process model* task). The engine-dependent test cases describe the behavior of the test partners. According to this specified behavior, if necessary, test partners are created and started in the *provide test partners* task.

Execute Phase

The *execute test cases* task then executes the engine-dependent test cases within the testbed containing the fresh and started test partners as well as the fresh engine with the process model deployed onto.

Teardown Phase

After the execution of the test cases, the collection of test case results are available and the teardown phase begins. First, the logs of both the test partners and the engine are collected and analyzed (*analyze log* task) to compute a more detailed collection of test case results, which can then be aggregated to the actual test result (*aggregate results* task). As part of the cleanup work in the teardown phase, the test partners and the engine have to be dispersed (*disperse test partners* and *disperse engine* tasks).

Engine

During the process, the engine is set into four states in the order described in Table 5.1. There is no task to uninstall or remove the engine. The execution of the task *provide engine* requires any previously running engine to already be shut down and uninstalled. Otherwise, the engine instance would not be fresh.

Mapping

These states of the engine and the tasks that manipulate them in PEBWORK are similar to the ones of PEAL. As a result, some tasks of PEBWORK can

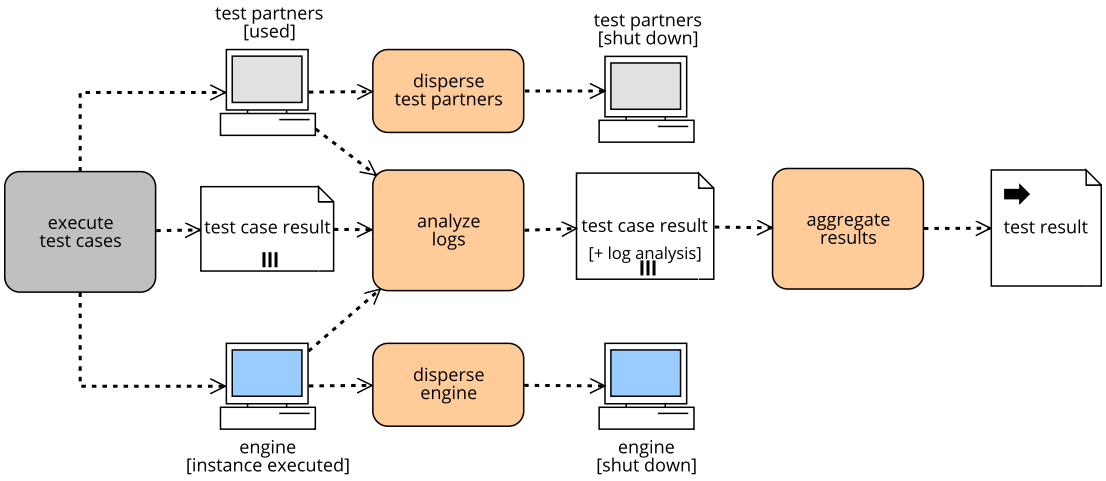


Figure 5.4.: Data Flow of the Benchmark Framework of the Engine Selection Process – Insider View of the Teardown Phase

Table 5.1.: Different States of the Engine Under Test

state	**description**
fresh and started	Engine is installed and it has been fully started. It was not used before in any other test, hence it is fresh. It is responsive to all commands. Log files are available documenting the current execution of the engine.
process model deployed	Same as *fresh and started* state, but a single process has been deployed to the instance. This allows starting instances of this deployed process.
instances executed	Same as *process model deployed* state, but one or more instances of the previously deployed process have been executed.
dispersed	Engine is not running anymore, and therefore unresponsive to any commands. The logs may still be available, but not necessarily. Dispersed can also mean that the engine is already uninstalled and completely deleted.

be implemented using operations defined by the uniform API of PEAL. A mapping of those is given in pseudo code in Listing 5.1. The *provide engine*, *create deployable unit*, *deploy process*, and *disperse engine* tasks can be directly mapped to one or more operations of the engine or process model service. The remaining two ones, being the *execute test cases* and the *analyze logs* tasks, use the instance or engine service to fulfill parts of their responsibilities.

Listing 5.1: Mapping of Benchmark Procedure Tasks to PEAL API Calls.

```
1 task provide engine:
2     call engineService.stop engineId
3     call engineService.uninstall engineId
4     call engineService.install engineId
5     call engineService.start engineId
```

```
6
7  task create deployable unit:
8      call processModelService.makeDeployable engineId, processModelPackage
9
10 task deploy process:
11     call processModelService.deploy engineId, deploymentPackage
12
13 task execute test cases:
14     // only for BPMN engines
15     call instanceService.start processModelId, bpmnVariables
16
17 task analyze logs:
18     call engineService.getLogs engineId
19
20 task disperse engine:
21     call engineService.stop engineId
```

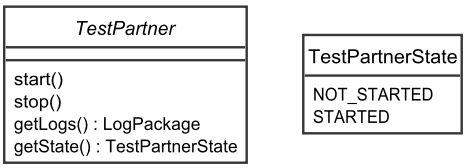


Figure 5.5.: The Lifecycle and Interface of the Test Partner

Test Partner

The other IT system in this process is a collection of test partners. The test partners simulate other components the process may interact with (i.e., communication partners within the test). They are also used to verify specific features (e.g., for Concurrency Detection (P16)). A test partner has to implement and conform to a simple interface as shown in Figure 5.5. As it is depicted, a test partner can either be *STARTED* or *NOT_STARTED*, which is triggered through the *start* and *stop* operations. If the test partner is in its *STARTED* state, the logs can be accessed as well, reusing the log package structure from PEAL (see Section 3.3.1). After its start, a test partner is in a fresh state until it is first used. To guarantee test isolation, a test partner needs to be set in a fresh state similar to an engine. In the BPMN data flow process, the state of the used test partners is modeled explicitly. An overview of the three different states within this process is summarized in Table 5.2.

Table 5.2.: Different States of the Test Partners

state	**description**
fresh and started	The test partners are set up and running. They have not been used before in any other test, hence they are fresh. Log files are available documenting the current execution of the test partners.
used	Same as *fresh and started* state, but test partners have already been used within one or more test cases.
dispersed	Test partners are not running anymore, and therefore unresponsive to any commands. The logs may still be available, but not necessarily. Dispersed can also mean that the engine is already uninstalled and completely deleted.

5.4. Prototype betsy

betsy

The prototype of PEBWORK is called BPEL/BPMN Engine Test System (betsy) and written in Java 8 and Groovy 2.4. The source code is publicly available[60] along with the Docker container in which betsy is already set up[61].

Overview

This section is structured as follows. First, the control-flow of the benchmarking procedure is detailed in Section 5.4.1. Second, the architecture implementing the previously described procedure and its limitations are outlined in Section 5.4.2 and Section 5.4.3, respectively.

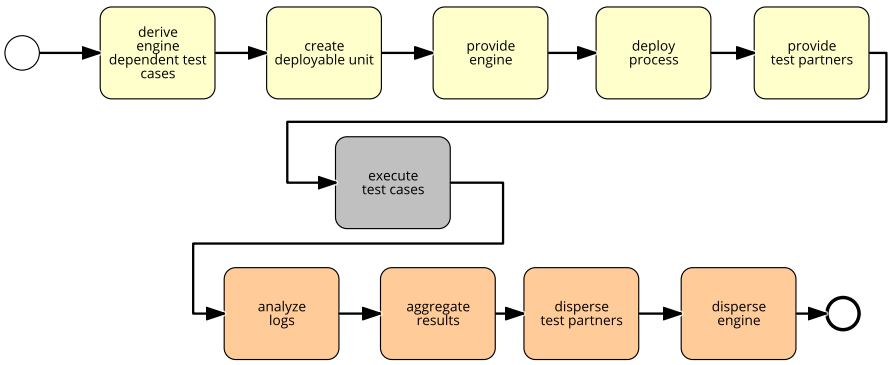


Figure 5.6.: Sequential Control-Flow Diagram of the Data Flow of the Insider View on the Benchmarking Framework

5.4.1. Control Flow

Sequential Control Flow

This prototype uses the sequential control flow as shown in Figure 5.6 because it is the most straightforward way to realize a prototype of PEBWORK. For clarity reasons, we have hidden all the *DataObjects* and *IT Systems*. One can easily recognize the three different phases in the control flow: the setup phase colored yellow, the execute phase gray, and the teardown phase orange. A variation of this sequential control flow diagram has already been published in earlier versions of this work [81, 95, 96]. In these publications, we used the sequential approach as it allows for a simpler implementation. Nevertheless, there is potential for parallel execution of the tasks in both, the setup and the teardown phase. Figure 5.7 shows the parallel version revealing the opportunities for parallelization separately for the setup and the teardown phase. In both phases up to three branches can be performed in parallel. The *execute test cases* task in between the setup and the teardown phase would act as a synchronization barrier in case of parallelization.

[60] `https://github.com/uniba-dsg/betsy`, visited 2017-3-31
[61] `https://hub.docker.com/r/simonharrer/betsy-docker/`, visited 2017-3-31

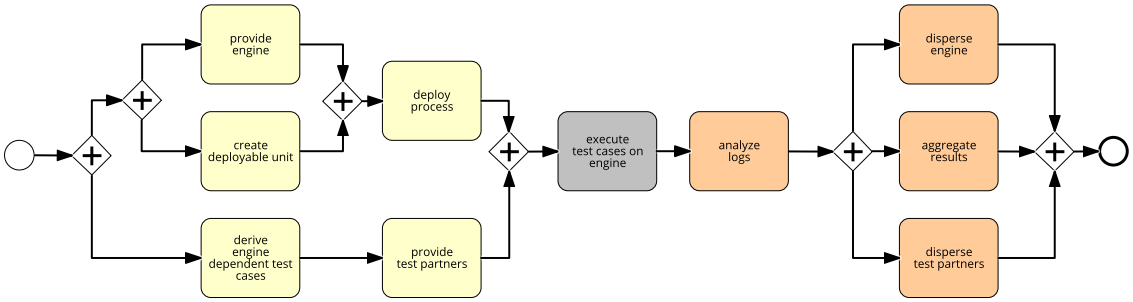


Figure 5.7.: Parallel Control-Flow Diagram of the Data Flow of the Insider View on the Benchmarking Framework

5.4.2. Architecture

Dependencies

The prototype betsy implements the sequential control flow that is depicted in Figure 5.6. It builds upon other software components as shown in Figure 5.8. Those are either internal (i.e., originated as part of this work) or external (i.e., developed and open sourced from someone else), and are detailed in the following.

External Dependencies

First, betsy relies on the PEAL prototype from Section 3.4 to interact with the process engines in a uniform way and the prototype of PEBL from Section 4.4 to read and write serializations of PEBL. Second, betsy itself uses several Apache Ant tasks to interact with files on the file system or executing commands on the console. This prototype relies on Apache Ant 1.9.2 and its corresponding Groovy bridge that is shipped as part of the Groovy language. Third, the task to *execute test cases* is conducted differently for BPEL and BPMN tests. For the BPEL tests, betsy relies completely on soapUI 4.6.4 for test execution. To achieve this, the test cases and their steps and assertions defined in PEBL are mapped to soapUI test cases using the soapUI-based XML format, and these soapUI test cases are then executed natively in soapUI, producing a JUnit XML report. In contrast, for BPMN test cases, the steps are executed within betsy, but the trace assertions are executed through JUnit 4.12 via an Apache Ant task, resulting in a JUnit XML report as well. Although the execution of BPEL and BPMN test cases is different, they both produce JUnit XML reports which can be further processed in the same way.

Internal Dependencies

Internally, betsy is structured into separate modules that depend upon or extend one another. The entry point is a *CLI* which receives the parameters from the user, and with the help of the PEBL prototype reads and writes the input and the output of betsy. It also triggers the test execution through the *executor* (i.e., the heart of betsy). The executor is a direct implementation of the sequential control flow from Section 5.4.1. It orchestrates these steps by calling the appropriate *tasks* that make use of Apache Ant or the API of PEAL to interact with the engines under test. *Timeouts* are necessary to wait for success or failure conditions when working with the slow file system or interacting

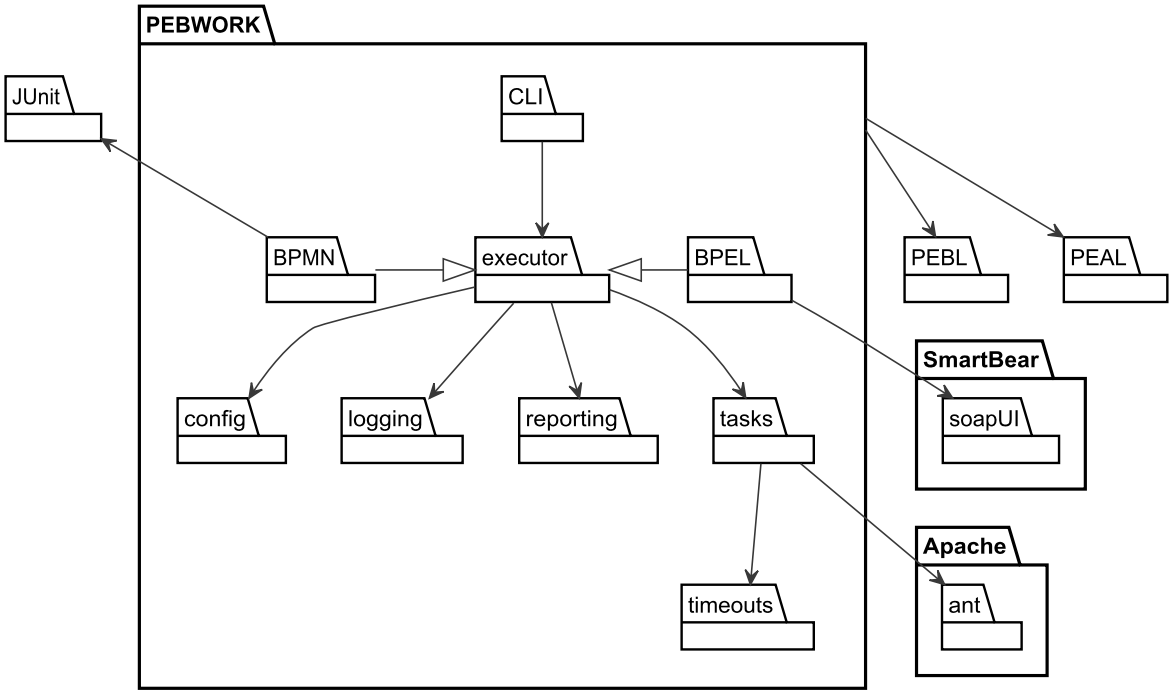


Figure 5.8.: The Architecture of betsy, the Prototype of PEBWORK

with other software systems through the file system. The executor can be configured through the *config* module. Insights into the executor at runtime are provided through *logging*, and afterward through *reporting* facilities. There is an *executor* which works independent of any process language, but there are process language specific extensions for *BPEL* and *BPMN* to handle the language specific issues.

5.4.3. Limitations

The prototype comes with some limitations. First, betsy inherits the limitations of the prototypes of both, PEAL and PEBL as it heavily relies on them. This includes limited deployment descriptor generation, no engine installation configuration options, limited ability to undeploy a process model, and only an instance service for BPMN engines (see Section 3.4.4). Second, the prototype can only perform benchmarks using the predefined test steps and assertions. Although it is possible to extend PEBL through user-defined scripts, the prototype does not interpret these. In all cases where scripts are necessary, the prototype provides a hard coded solution based on the naming of the elements with the user-defined scripts. Third, betsy cannot perform performance benchmarks. This is simply not supported and would require a major extension as in that case the environment has to be more complex because one normally requires different isolated environments for generating the workload, the engine, and its database. Fourth, conducting large benchmarks on many engines takes

some time as betsy is not optimized for efficiency. For instance, betsy executes everything sequentially, only caches downloads and not any computed artifacts, and relies on Reinstallation (P9) to ensure test isolation.

5.5. Evaluation

Practical and Theoretical Evaluations

The evaluation comprises two prototypical and one theoretical evaluation. First, the prototype of PEBWORK is evaluated through an extension of the aptitude test that has been performed (see Section 3.5) to determine that betsy is effective in the simplest benchmark possible (see Section 5.5.1). Second, betsy is evaluated by performing the benchmarks with which PEBL has been evaluated in Section 4.5. This determines whether betsy can also perform more complex benchmarks (see Section 5.5.2). Third, it is evaluated whether the triplet of PEBWORK, PEBL, and PEAL is is capable of enabling good benchmarks according to the criteria summarized in Section 2.3.2 (see Section 5.5.3).

5.5.1. Prototypical Evaluation: Aptitude Test

Method

The first evaluation builds upon and extends the evaluation of PEAL in Section 3.5 to an evaluation of PEBWORK. The setup and the process models for both, BPEL and BPMN are re-used and only the evaluation procedure differs. Instead of calling the PEAL API directly, the evaluation only issues a command to betsy and determines whether the resulting PEBL serialization contains the expected results. Hence, it is called the Aptitude Test (P8) as it determines whether betsy can perform a test on a specific engine. Each available engine from PEAL is part of this evaluation so that the aptitude of betsy to test all the supported engines from PEAL is evaluated. The evaluation is also encoded as JUnit tests[62] and uses the PEBL serialization of the conformance tests for the features BPEL *Sequence* and the BPMN *SequenceFlow*. In the serialization of PEBL, everything is encoded to reproduce the test, fulfilling requirement R5.1.

Results & Discussion

The results[63] show that betsy succeeds for both, BPEL and BPMN engines supported by PEAL on both, Windows and Linux. This is supported as the actual logs are provided as well, which allow double checking if necessary, fulfilling requirement R5.3. In other words, betsy is in general able to perform benchmarks defined in PEBL and engines implementing PEAL through following the steps defined by PEBWORK on multiple platforms. Hence, as this is fully automated, requirement R5.4 is fulfilled. There is, however, the issue of efficiency in general, because the test duration varies greatly for this Aptitude Test (P8), ranging from three up to ten minutes per test. This is attributed to

[62] `https://github.com/uniba-dsg/betsy/blob/master/src/test/groovy/systemtests/BPELSystemTest.java` and `https://github.com/uniba-dsg/betsy/blob/master/src/test/groovy/systemtests/BPMNSystemTest.java`, visited 2017-3-31.

[63] See the travis builds at `https://travis-ci.org/uniba-dsg/betsy`, visited 2017-3-31

the costs of providing an isolated test environment through Reinstallation (P9), which is required to fulfill requirement R5.2.

5.5.2. Prototypical Evaluation: Case Studies

Method The second practical evaluation extends the first one by performing the benchmarks defined in Section 4.5, except the performance benchmark, because betsy cannot perform performance benchmarking. This evaluation aims to show that betsy as the implementation of PEBWORK can benchmark along the three axes: different process engines, different process languages, and different engine capabilities.

Results The results of this evaluation have already been published. Table 5.3 lists which capability results are published in what publication for each process language. All of them are peer-reviewed and either publications in conference proceedings or journals that are well known in this field of research.

Table 5.3.: Published Benchmark Results per Capability

Engine Capability	**BPEL**	**BPMN**
Conformance	[95]	[85]
Static Analysis	[99]	[85]
Expressiveness	[96]	[85]
Robustness	[98]	

Status Quo The evaluations in any of these publications are snapshots at different points in time. Since then, new versions of the engines under test have been released, errors in the benchmarks found, the benchmarks extended, and the benchmark tools improved. These evaluations show that PEBWORK is a suitable procedure to determine high-quality benchmark results. An update to the newest engine versions, improved benchmarks, or even newer versions of betsy will not have any significant impact on the already uncovered findings. Any state of "up-to-date" elements will quickly be out of date when new versions of the engines under test are released or more extensive benchmarks are published and created. Consequently, in this work, only the major findings and take away points from these already published evaluations are stated, instead of redoing these evaluations with the most up-to-date versions which are simply outdated some while later. For a more detailed evaluation, please have a look at the corresponding publications given in Table 5.3.

At Least One Success (P21) The At Least One Success (P21) pattern is applied to ensure that the results are correct. This pattern works by determining whether a test has been passed by at least a single engine. If this is the case, this indicates that the process is correct, and if not, the process is probably erroneous. 95% of the tests are passed by at least one engine [95].

The time to results, however, is high. A run with 130 tests for five BPEL engines took approximately ten hours to complete [95]. That is about one minute per test on a single engine on average. Executing the whole evaluation with its 1223 BPEL tests on 20 BPEL engines and 447 BPMN tests on 22 BPMN engines would take approximately 34294 minutes which corresponds to 571.6 hours or 23.8 days. A single BPEL or BPMN engine, however, can be evaluated in 20.4 hours and 7.5 hours, respectively. This is manageable for evaluating engines for a selection decision, but not as part of a CI pipeline that is run for each checked in change. Hence, this acts as the motivation for the Efficient Process Engine Benchmark Framework (ePEBWORK), which is a more efficient version of PEBWORK. Time to Results

5.5.2.1. BPEL Results Summary

An abbreviated overview of the results from the four benchmarks is given in Table 5.4. It visualizes the differences between engines. The engine bpel-g comes in first three times, with Apache ODE being on par once regarding the expressiveness. OpenESB detects injected faults better than any other tested engine. In the following, the highlights of each BPEL benchmark are stated. The winner per benchmark is marked via the symbol 🏆. Summary

Table 5.4.: Results for BPEL Benchmarks for the Evaluated Engines [96, 98, 99]

Process Engine	**Conformance** features in %	**Expressiveness** patterns in %	**Static Analysis** rules in %	**Robustness** faults in %
Apache ODE 1.3.5	66	🏆 100	30	0
bpel-g 5.3	🏆 92	🏆 100	🏆 75	90
OpenESB 2.2	66	63	0	🏆 94
Orchestra 4.9	50	69	54	18
Petals ESB 4.0	26	56	4	0

Language feature support varies greatly between the benchmarked BPEL engines, ranging from 26% up to 92%. Hence, there is no engine that supports the whole standard, but bpel-g supports the vast majority of the features. Nevertheless, even only a quarter of the features supported by Petals ESB are seen as a set of features that can be put to use. Furthermore, the features that are less supported are the *flow* activity with *links* and the *forEach* activity. Additionally, these two activities can be set to be executed in parallel, but in practice, the *forEach* activity is executed sequentially. Conformance

The support for detecting erroneous process models differs greatly as well, ranging from no support at all up to 75% of the evaluated 71 static analysis rules. The rules detecting control cycles are only supported with an average of 25%. This, however, is problematic as this detection is crucial to prevent erroneous *flow* with *links* configuration upfront. Static Analysis

Expressiveness

The expressiveness benchmark shows that at least half of the patterns are supported by each benchmarked engine, even with only support for a quarter of the language features. Moreover, with solely 66% of the language features, Apache ODE can support all workflow patterns.

Robustness

The benchmark revealed that there are three different approaches on how to handle faulty incoming messages: throw a fault within the process instance (for bpel-g and OpenESB), do not route the message to the process instance (in Apache ODE and Petals ESB), and ignore any faults and simply route the message as is to the process instance (in Orchestra). The usage of the BPEL activity *validate* can help to improve robustness, but it only had an effect for OpenESB, moving it up six percentage points to the leader in that benchmark, as shown in Table 5.4. Although OpenESB, admittedly, has the best runtime fault detection, it lacks any compile-time detection capabilities [98].

5.5.2.2. BPMN Results Summary

Summary

In summary, each engine is the winner in one of the three benchmarks as visualized in Table 5.5. This, again, shows the importance of benchmarking the quality characteristics of the engines. In the following, the highlights of each BPMN benchmark are stated.

Table 5.5.: Results for BPMN Benchmarks for the Evaluated Engines [85]

Process Engine	**Conformance** features in %	**Expressiveness** patterns in %	**Static Analysis** rule configs in %
activiti 5.20.0	46	🏆 83	50
jBPM 6.4.0	53	72	🏆 78
camunda BPM 7.5.0	🏆 57	78	47

Conformance

From the three evaluated BPMN engines, the supported language features range between 48 and 60 out of 105. Although the degree of support does not vary that much, it is approximately half of the evaluated features. This shows that many BPMN elements cannot be used for process execution. Furthermore, no real parallelism is supported, but only pseudo parallelism [85].

Static Analysis

The ability to detect erroneous process models through static analysis varies greatly among the three BPMN engines, ranging from 141 up to 236 out of 301 rule configurations. The engine jBPM 6.4.0 has the highest detection rate of approximately 78%, and camunda BPM 7.5.0 the lowest with only approximately 47%. This leaves a lot to be improved [85].

Expressiveness

From the 18 patterns supported by BPMN, between 13 and 15 are supported by the three evaluated engines. The engine *activiti 5.20.0* supports 15 patterns with less than half of the evaluated construct configurations, whereas *jBPM 6.4.0* supports solely 13 patterns with eight more construct configurations than

activiti. Hence, the support is high, and the differences are only minimal between the engines [85].

5.5.3. Theoretical Evaluation: Good Benchmarks

In the theoretical evaluation, it is evaluated whether the usage of PEBWORK helps in creating good benchmarks. If PEBWORK helps in creating good benchmarks, the usage of the framework has inherent value: it would enable good benchmarks with lesser effort than before. In Section 2.3.2, a good benchmark has already been defined [111, 229], and the criteria of a good benchmark have been condensed to nine elements: *affordable*, *relevant*, *portable*, *accessible*, *clear*, *solvable*, *scalable*, *repeatable*, and *verifiable* (see Table 2.6). In the following, it is checked whether the framework itself and through the usage of PEAL and PEBL ensures that one or more of those criteria are fulfilled inherently through its usage. An overview is given in Table 5.6. Method

Table 5.6.: Theoretical Evaluation of PEBWORK with details about PEAL and PEBL according to the Good Benchmark Criteria

Criteria	**PEAL**	**PEBL**	**PEBWORK**	Σ
Affordable	engine mapping	DSL	automation	+
Relevant				
Portable	engine mapping	standards	any OS	+
Accessible		aggregation	open source	+
Clear		DSL		+
Solvable				
Scalable	engine mapping			+
Repeatable	API	DSL	isolation, automation	+
Verifiable	aptitude test	static analysis	traceability	+

The abstraction layer PEAL makes the benchmark more affordable, portable, scalable, repeatable, and verifiable. Because of its engine mapping, it is easier to integrate an additional engine, such as a new version of an existing engine (affordability). The mapping does not require a lot of methods to be mapped, hence, any product that simply implements this can be added (scalability and portability). The API of PEAL is the foundation of a repeatable benchmark because it enables Reinstallation (P9) and provides a uniform single point of access for the required engine functionality (repeatability). Whether an engine fits can be determined through the Aptitude Test (P8) (verifiability). PEAL

The benchmark language PEBL makes the benchmark more affordable, portable, accessible, clear, repeatable, and verifiable. The DSL of PEBL reduces the effort to define a benchmark as it provides the necessary elements to express a benchmark (affordability). Expressing the benchmarks in that DSL ensures that the benchmark is described in a way so that it can be executed over and over again (repeatability). The DSL contains domain-specific elements, and PEBL

therefore the representation of the benchmark is concise and clear (clarity). With PEBL, benchmarks can be defined for the two process language standards BPMN and BPEL. The benchmarks themselves, however, do not require any engine-specific information. Hence, benchmarks can be performed for any engine that supports the appropriate process language standard (portability). Through the definition of a metrics aggregation hierarchy, the results are much easier to compare (accessibility). And last, because the prototype of PEBL makes use of BPELlint and BPMNspector to ensure that the process models are correct, the benchmarks themselves are checked (verifiability).

PEBWORK The benchmarking framework PEBWORK itself makes a benchmark more affordable, portable, accessible, repeatable, and verifiable. The most important aspect of PEBWORK is that it ensures test automation and test isolation (repeatability). If PEBWORK itself is executed within a Docker container, the runtime environment is fixed as well ensuring even higher reproducibility. Moreover, the benchmarking framework allows performing benchmarks with standard hardware and minor development overhead (affordability). Because PEBWORK is open source, there is no entry-barrier to not use the prototype betsy. Also, a preconfigured runtime environment of PEBWORK and all the benchmarks are available as a Docker container (accessibility). The prototype also runs on both, Windows and Linux (portability). And last, every step that is made within betsy is reported and all the engine-specific logs are captured to be able to trace and check any flaw or issue in the result back to its cause (verifiability).

Summary To sum up, PEBWORK is useful because it enables a good benchmark as demonstrated in Table 5.6. The triplet of PEAL, PEBL, and PEBWORK help to fulfill seven out of the nine criteria better. The two criteria relevance and solveability are not covered by them because they are solely dependent on the actual benchmark and cannot be influenced by such contributions as the three in this work.

5.6. Summary

Summary In this chapter, the Process Engine Benchmark Framework (PEBWORK) has been presented that allows conducting benchmarks that are specified in Process Engine Benchmark Language (PEBL). The fully automated prototype BPEL/BPMN Engine Test System (betsy) implements this approach. The two prototypical evaluations proved its effectiveness and efficiency because the derived requirements for such a framework are fulfilled. That PEBWORK enables good benchmarks is proven in the theoretical evaluation: it fulfills seven out of nine criteria of a good benchmark through its inherent design. To sum up, the results of this evaluation supports hypothesis H4.3 *(“A benchmarking framework is a suitable means to reveal objective and well-founded information about process engines.”)*.

Efficiency

The approach PEBWORK and the prototype betsy are efficient to some degree by being fully automated but there is a lot of potential, still. Because of this, a more efficient version of PEBWORK is detailed in Chapter 7 that makes use of virtualization technologies: Efficient Process Engine Benchmark Framework (ePEBWORK).

Future Work

Additional future work comprises two aspects: evaluation and implementation. Currently, the evaluation has been made with the benchmarks and engines that have been available as part of PEBL and PEAL. In the future, PEBWORK will be evaluated more thoroughly with additional benchmarks for other capabilities and quality characteristics (e.g., performance) which come with additional requirements and challenges regarding test automation and test isolation. The implementation, currently, is limited as well due to the limits of PEBL and PEAL. Hence, the goal is to remove those limitations to improve betsy itself.

Data! Data! Data! I can't make bricks without clay!

Sir Arthur Conan Doyle

6. Process Engine Benchmarking Interactive Dashboard

Parts of this chapter have been taken from [18].

In this chapter, hypothesis H4.4 *("An interactive dashboard is a suitable form to present benchmarking results and support selection decisions.")* is supported.

This chapter presents the Process Engine Benchmarking Interactive Dashboard (PEBDASH). It is used to display and visualize the benchmarks and results defined in the Process Engine Benchmark Language (PEBL) (see Chapter 4) and produced by the Process Engine Benchmark Framework (PEBWORK) (see Chapter 5).

6.1. Motivation

Benchmarks are often used by end-users to evaluate and compare competing systems before deciding on the system to adopt [192]. Likewise, researchers and developers use benchmarking techniques to evaluate the design and improvement of systems over time. In the field of process engines, benchmarks for testing different capabilities of process engines have been built [61, 81, 95], one being this work as well. These benchmarks may support users, developers, and researchers in analyzing and comparing process engines for BPMN and BPEL. The results published so far, however, only show aggregated data of one or two engine capabilities to get their point across. The actual raw data is sometimes published as well but uses different data formats and levels of detail. Hence, working with this raw data is either not possible at all or hard. This, however, is crucial since the raw data would allow gaining additional insights, e.g., a clear picture of different capabilities per engine. As with many benchmarks (e.g., for databases or Web servers), the benchmarks for process engines only show raw data with few aggregation and less emphasis on visualization and interaction with the data. Although raw data and noninteractive result tables are sufficient and even preferred by expert users, novice end-users may find it difficult and time-consuming to choose their best-fitting process engine with this level of detail [294, p. 52]. In contrast to experts, novices may have not enough

knowledge to analyze, compare, and identify problems using atomic raw data. Also, raw data may not meet the different cognitive skills (e.g., analytical versus less analytical) [294, p. 49] of end-users. Yigitbasioglu and Velcu [294] have shown that an appropriate aggregation and visualization of data helps users in understanding complex information, identifying problems and focusing on the most relevant data (e.g., metrics) for better decision making. Furthermore, at the moment, benchmark data for process engines are not publicly available or accessible. This makes it difficult for end-users to make informed decisions about choosing an appropriate process engine without benchmarking these engines themselves.

Goals of Dashboard

To overcome the limitations above, a web-based interactive dashboard is created that makes process engine benchmarks accessible, visible, and comprehensible to end-users. In general, the purpose of dashboards is to facilitate the understanding of data and metrics by making use of colors, tables, charts, and interactive techniques (e.g., data filtering or drill-down) in the visualization of this data. By building a dashboard, the focus mainly is on empowering non-expert end-users to understand, analyze, and access benchmark results easily. Nevertheless, experts may still use the dashboard to manage their benchmarks, to compare them with the results presented in the dashboard, or to update test implementations. This could establish a community of discussing and sharing engine benchmarks. Further, when using a dashboard, experts may profit from the more aggregated and structured presentation of the benchmark results to communicate them to nonexperts stakeholders effectively [196]. In summary, the hypothesis H4.4 *("An interactive dashboard is a suitable form to present benchmarking results and support selection decisions.")* is proposed.

Chapter Structure

The remainder of this chapter presents the design and implementation of the Process Engine Benchmarking Interactive Dashboard (PEBDASH) for comparing process engines by visualizing the benchmarks and their results written in PEBL. First, it is described what dashboards are and how they are used in Section 6.2, followed by the listing of requirements the interactive dashboard shall fulfill in Section 6.3. Next, Section 6.4 describes the approach on implementing the requirements in the form of an interactive dashboard and a loader which loads benchmark runs into the database of the dashboard. The actual implementation of the two artifacts is presented in Section 6.5, followed by an evaluation in Section 6.6 based on the previously defined requirements. Finally, Section 6.7 sums up this chapter.

6.2. Dashboards

The term *dashboard* has its origin in the vehicle dashboard which displays key performance indicators (e.g., speed or fuel gauge) of the vehicle [294]. Few [64] provides a definition of a dashboard.

Definition 6.1 (Dashboard)
"A dashboard is a visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance." [64]

Dashboard

Based on this definition, in the context of this thesis, a dashboard is understood as a tool to collect, summarize, and present a large set of benchmark results from multiple sources over time on web pages so that key performance indicators of benchmarks can easily be perceived and understood at a glance. In this work, a web-based dashboard for benchmarks consists of web pages displaying most important information about the results of different benchmarks (e.g., conformance, expressiveness, and performance as presented in Section 4.5). Although dashboards are commonly used as part of Business Intelligence (BI) systems for measuring and monitoring business performance to support managerial decision making [294], their characteristics and design principles have been adopted for reporting benchmark results.

Web-Based Dashboard

In the following, the common characteristics and advantages of web-based dashboards for visualizing benchmark results are highlighted. Benchmark dashboards typically collect and summarize data from different benchmark runs to allow the analysis and the comparison of these benchmark runs. Some dashboards collect benchmark results of the same quality characteristic [118] over a period, which may be helpful for identifying trends such as the stability or the improvement of the tested system. A web-based dashboard is a method for making a large set of benchmark data available in one place on the web, and thus easy to find, access, and share [14]. When done right, dashboards display few metrics and aggregated information to reduce the information load, and to help users quickly get an overview of the benchmark results. Interactive dashboards presenting aggregate data mostly allow users to navigate from highly aggregated data to the more detailed level of data, e.g., to obtain additional details on a particular benchmark run. This feature is known as drill-down [196]. Furthermore, interactive dashboards often provide a mechanism for filtering data to find and compare specific parts of the benchmark results quickly. More importantly, filters allow the exploration of data which is useful for users unfamiliar with the data [57]. One important characteristic of dashboards is the focus on making the visual presentation of data intuitive and easy to understand. This is typically achieved by making use of colors, distinctive signs (e.g., $+$, $-$, or $+/-$), charts, and graphs. Those ease the identification of key metrics and the comparisons of the results.

Available Dashboards for Benchmarks

The characteristics and design principles of dashboards have been adopted in a variety of web-based dashboards for reporting benchmark results. For example, TechEmpower Framework Benchmarks (TFB)[64] is a dashboard that presents performance benchmark results of web frameworks to facilitate the

[64] `https://www.techempower.com/benchmarks`, visited 2017-3-31

comparison of these frameworks regarding their performance. TFB has become a community-driven project in which contributors frequently update test implementations. Compat-table[65] is a dashboard that presents benchmark results that evaluate the compatibility of browsers, servers, and compilers with ECMAScript 5, 6, and 7. The Node-compat-table[66] dashboard only focuses on evaluating different versions of the Node.js server regarding ECMAScript support. Both dashboards make extensive use of colors to simplify the differentiation and the recognition of the result ratings. Also, they employ drill-down features to enable aggregation of the ECMAScript features. JMeter[67] is a performance testing framework that comes with a dashboard for displaying the benchmark results using tables and graphs. It provides filters to specify which rows of tables or series of graphs should be shown or hidden. Another benchmark framework that provides a dashboard is Rally[68]. It is used for benchmarking the OpenStack[69] cloud hosting infrastructure. The Rally dashboard simplifies the collection of test results and extracts relevant metrics to help users improving their OpenStack infrastructure. Similar to PEBDASH, the Rally dashboard also provides information about the configuration of each test (e.g., test cases or configurations). Another dashboard or system that helps in selecting the appropriate benchmark for process engines is the Decision Support System (DSS) for the performance benchmarking of process engines by Skouradaki et al. [236]. Its focus, however, is not on selecting the appropriate engine but on selecting the appropriate performance benchmark. The ultimate IoT platform comparison[70] provides a dashboard for comparing different IoT platforms, obviously. Kolb and Wirtz [133] presents a dashboard[71] to compare different PaaS solutions by different vendors.

Research Gap

Despite the existence of a variety of web dashboards for benchmarking tools, there is no *interactive* dashboard yet for presenting the benchmarking results of process engines. Only the presentation of the workflow patterns dashboard [291] has few characteristics of a dashboard, but it solely displays result tables with no interactivity at a high level of granularity.

[65] `http://kangax.github.io/compat-table`, visited 2017-3-31
[66] `http://node.green`, visited 2017-3-31
[67] `http://jmeter.apache.org/usermanual/generating-dashboard.html`, visited 2017-3-31
[68] `https://www.mirantis.com/blog/rally-as-an-openstack-performance-dashboard/`, visited 2017-3-31
[69] `https://www.openstack.org/`, visited 2017-3-31
[70] `https://ultimate-comparisons.github.io/ultimate-IoT-platform-comparison/`, visited 2017-3-31
[71] `https://paasfinder.org`, visited 2017-3-31

6.3. Requirements

This section discusses the requirements for the dashboard. The requirements were gathered by leveraging user stories popularized by the agile software method Scrum [209, 225].

Stakeholders

There are three different stakeholders, namely, the *user* representing anyone who either wants to use or already uses a process engine, the *vendor* representing anyone who develops, sells, or provides support for an engine, and the *admin* who maintains the dashboard. The user is looking for a process engine to use and wants to know which one fulfills his needs best and which one would lead to the least danger of a vendor lock-in. It, however, could also be the case that the user already uses a process engine and is thinking about migrating to another one and wants to know which engine supports the most of the required features. A special case would be the migration from an older version of a process engine to a newer one. In contrast, the vendor is looking forward to knowing how his product compares to its competitors and how his product has evolved over time feature-wise. Last, the admin wants to minimize his effort to host and maintain the dashboard.

Requirement Elicitation

The requirements were elicited by analyzing the benchmark results of both betsy and BenchFlow, by questioning domain experts from different universities within Europe (e.g., University of Bamberg, University of Stuttgart, Karlstadt University, and University of Lugano), and by reviewing scientific papers and existing web dashboards. The captured requirements were decomposed into user stories [225] and validated by letting the experts try out an interactive prototype upon which the most important Functional Requirements (FRs) and Nonfunctional Requirements (NFRs) for the dashboard are gathered:

FR.User.1 – compare all aggregated: As a user,
I want to see aggregated information on all engines in one place
so that I can quickly gain an overview of all engines as an entry point for further investigations.

FR.User.2 – compare latest feature: As a user,
I want to see differences and similarities in feature support of the latest version of all engines
so that I can make an informed decision regarding which engine to choose.

FR.User.3 – compare latest aggregated: As a user,
I want to see differences and similarities in feature support on an aggregation level of the latest engines
so that I can make an informed decision which engine to choose more quickly.

FR.User.4 – compare versions feature: As a user,
I want to see differences and similarities in feature support of multiple

versions and configurations of the same engine
so that I can check if an upgrade is worthwhile.

FR.User.5 – drill down to feature: As a user,
I want to see all features for each (unsuccessful) aggregation level
so that I can check if all features are unsupported or if there are problems with some of the features in this aggregation.

FR.User.6 – compare any feature: As a user,
I want to see differences and similarities in feature support among different engines
so that I can estimate the migration costs between engines (see vendor lock-in).

FR.Vendor.1 – compare latest: As a vendor,
I want to compare the strengths and weaknesses of the latest version of all engines
so that I can know how my engine compares to its competitors.

FR.Vendor.2 – compare engine progress As a vendor,
I want to compare multiple versions of the same engine
so that I can reveal the feature progress of the engine.

FR.Vendor.3 – drill down to failure: As a vendor,
I want to be able to drill down to the cause of any failed feature on my engine
so that I can verify and fix the failure.

NFR.1 – hosting: As a admin,
I want to avoid any server-side logic
so that I can keep hosting and maintenance simple and secure.

NFR.2 – easy to use: As a user/vendor,
I want to have an easy to use and visually appealing dashboard
so that I can find the information I need quickly.

The FRs are subdivided into FRs for the user and the vendor. Both stakeholders have similar requirements which are phrased differently because of their perspectives (see *FR.Vendor.1/FR.User.2*, *FR.Vendor.2/FR.User.4*, and *FR.Vendor.3/FR.User.5*). The vendor wants to compare his product with the competition, reveal how his product improves over time, and quickly determine how to improve his product. The user wants to compare competing products, check whether the upgrade to the new version is worth the effort, and determine why a feature is not supported.

6.4. Approach

The big picture of the approach can be seen in Figure 6.1. Its goal is to fulfill the requirements listed in Section 6.3 . The approach consists of three software systems (benchmark tool, loader, and dashboard) and two persistent storages (run and database) connected through the load pipeline.

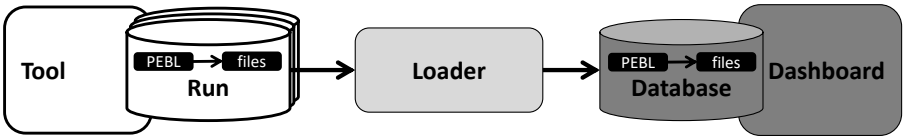


Figure 6.1.: Big Picture, including the Loader Pipeline

Tool On the left-hand side of Figure 6.1, there is a benchmark *tool* with which one can conduct benchmarks and produce benchmark results in benchmark *runs*. Such runs need to use a language and data model to express and serialize their benchmark and results to disk. In this approach, the runs need to use PEBL introduced in Chapter 4. As files can only be included in PEBL through links, they have to be provided as well so that the links can be followed. Each evaluation made in Section 4.5 represents a single run. Later, when a new engine or version of an existing is released, a new run has to be conducted for the new engine reusing the existing benchmark, but producing new results.

Dashboard On the right-hand side of Figure 6.1, there is the dashboard which displays the benchmarks and the results available in the database to the user. It is built with HTML5 and client-side only JavaScript fulfilling the ECMAScript 6 (ES6)[72] language specification. Hence, the dashboard can solely rely on the JSON serialization format of PEBL and does not require any server-side component except a simple web server servicing static files. The dashboard uses aggregated results described in Section 4.3 to display the relevant metrics to the user grouped by the engines. Also, it allows drilling down to the actual tests and logs. Moreover, features can be compared between engines and filtered to see only the features one is interested in.

Database The dashboard relies heavily on its database. Figure 6.2 shows the folder structure of the database. All the files are stored in the *data* folder next to the serialization of the *pebl.json* in appropriate subfolders. Because the IDs of the tests and the test results can become long, and Windows allows only 260 characters for the path name[73], they are hashed.

Loader In the middle of Figure 6.1, there is the loader. The loader is responsible for loading the results of a single run into the database of the dashboard. Through the arrows in Figure 6.1, the load pipeline is visualized. This load process also

[72] ES6 is the current standard for JavaScript and supports, among other features, modules natively.

[73] `https://msdn.microsoft.com/en-us/library/aa365247(VS.85).aspx#maxpath`, visited 2017-3-31

Figure 6.2.: Folder Structure of the Dashboard Database with File Examples

includes the enhancement and validation of the data. The pseudo code of the load process is available at Listing 6.1. The process comprises two parts and five steps in total. In the first part, data from the run is copied into the database. First, the new metric types, elements in the feature tree, tests, engines and test results from the PEBL serialization of the run are copied into the database. Second, any files that are linked from the database but are not within the folder structure of the database are copied to their expected place. In the second part, the data within the database is cleaned, improved, and validated. For each linked file that is a process model, an image is automatically created so that the user has a visual representation of the process models in addition to the XML-based serialization format. Also, aggregated results are recalculated to reflect the new elements. Moreover, any files in the database, which are no longer linked from the PEBL serialization of the database, are deleted. Last, the serialization is validated against its schema to prevent corruption of the database. In summary, the loader acts like an Extract-Transform-Load (ETL) process known from the field of BI that make use of Data Warehouses (DWHs) [127]. In contrast to a DWH in which data should be nonvolatile [127], the loader overwrites the results if a newer benchmark result is mapped to an existing one. This is useful if a better version of an existing test has been created.

Listing 6.1: Pseudo Code of the Loader Algorithm

```
1 def load(run, database):
2     from run into database:
3         copy/replace metric types, feature tree, tests, engines, and test
              results
4         copy/replace files of tests and test results
5     in database:
6         add images of process models
```

```
7         recalculate aggregated results
8         remove unlinked files
9         validate serialization
```

Scope

Not every benchmark tool supports PEBL. Some use a different language and data model to express their benchmark and results, as it is the case for BenchFlow [61, 237]. In these circumstances, the tool specific language and data model have to be mapped to PEBL. This must be done separately and is neither covered in the loader nor in this approach. For BenchFlow, this is done as part of the evaluation of PEBL in Section 4.5.3.4. Hence, this mapping can act as a starting point for other benchmarking tools.

6.5. Prototype

This section outlines the implementation of both, the loader in Section 6.5.1 and the dashboard in Section 6.5.2. The loader is implemented in Java 8, whereas the dashboard relies on HTML5 and ECMAScript 6 (ES6) alone. Both interact with each other through the database which is encoded in JSON.

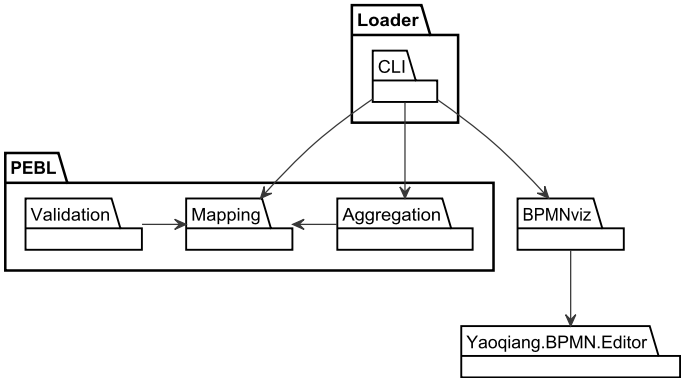


Figure 6.3.: The Dependencies of the Loader

6.5.1. Loader

Techno-logy

The *loader* loads a run into the database of the dashboard. The loader is usable through a CLI. It is open source and publicly available[74], including the documentation of the CLI[75]. This software is implemented in Java 8. Its dependencies can be seen in Figure 6.3. The loader uses the module *Mapping* and *Aggregation* of *PEBL* to read and write JSON serializations of PEBL as well as to compute aggregated results. A PNG file is automatically created for every BPMN process model via *BPMNviz* that is based upon the Open Source BPMN

[74] `https://github.com/uniba-dsg/betsy/tree/master/loader`, visited 2017-3-31
[75] `https://github.com/uniba-dsg/betsy/blob/master/loader/README.md`, visited 2017-3-31

Modeler *Yaoqiang BPMN Editor*[76]. The loader uses SHA-1 [55] as its hash algorithm, creating hashes which are 40 characters long for the folder structure as shown in Figure 6.2.

Limitations

The current prototype does not support the creation of PNG-based visualizations for BPEL process models. This is only implemented for BPMN process models. Moreover, as the loader uses the *Aggregation* module from the PEBL prototype, it inherits its limitations stated in Section 4.4. Namely, Groovy scripts are not supported.

6.5.2. Dashboard

Dashboard Pages

The dashboard is open source[77] and publicly available[78] as well. The interested user only needs a modern browser, such as Google Chrome 50 or Mozilla Firefox 49 or higher, to view the dashboard over the Internet. The dashboard contains seven pages: The *start page* provides links to the pages containing the actual benchmark results depicted in Figure 6.4. Three *capability pages* have been created, each of them presenting benchmark results of different capabilities: for *conformance* (Figure 6.5), *expressiveness* (Figure 6.6), and *performance* (Figure 6.7). The *engine compare* page (Figure 6.11) allows users to compare two different engines by their features and benchmark results. Note that only engines of the same process language can be compared. The *engine overview* page (Figure 6.10) allows the user to quickly get an overview of the engines and how each engine supports the different capabilities. Information about the project is included in the *about* page.

Extension

The dashboard has been built for extension. It can automatically adapt to new benchmarks and results as long as the structure of PEBL is used and extension points such as the custom names for groups, feature sets, and features as in Table 4.3 is leveraged, too. To show a new capability with its benchmarking results, it is necessary to provide the appropriate HTML templates and specify which of the result tables, filters, and drill-down functions have to be selected. Although some filters can automatically be provided, special filters that are dependent on specific metrics have to implemented as part of the extension.

Conformance Results

Figure 6.5 shows an example of conformance benchmarking results of the language BPMN. One can see the three levels of the feature tree for the conformance capability from Section 4.5.3.1, with the construct group *activities*, the construct *AdHocSubProcess*, and its two construct configurations *Parallel* and *Sequential*. The ratings in the columns display the metric trivalent aggregation.

Expressiveness Results

The presentation of the expressiveness benchmark results (Figure 6.6) is similar to that of the conformance benchmarking in the conformance capability page. This is because the benchmarks are similar (see the benchmark definition

[76] `http://bpmn.sourceforge.net/`, visited 2017-3-31
[77] `https://github.com/peace-project/dashboard`, visited 2017-3-31
[78] `https://peace-project.github.io/`, visited 2017-3-31

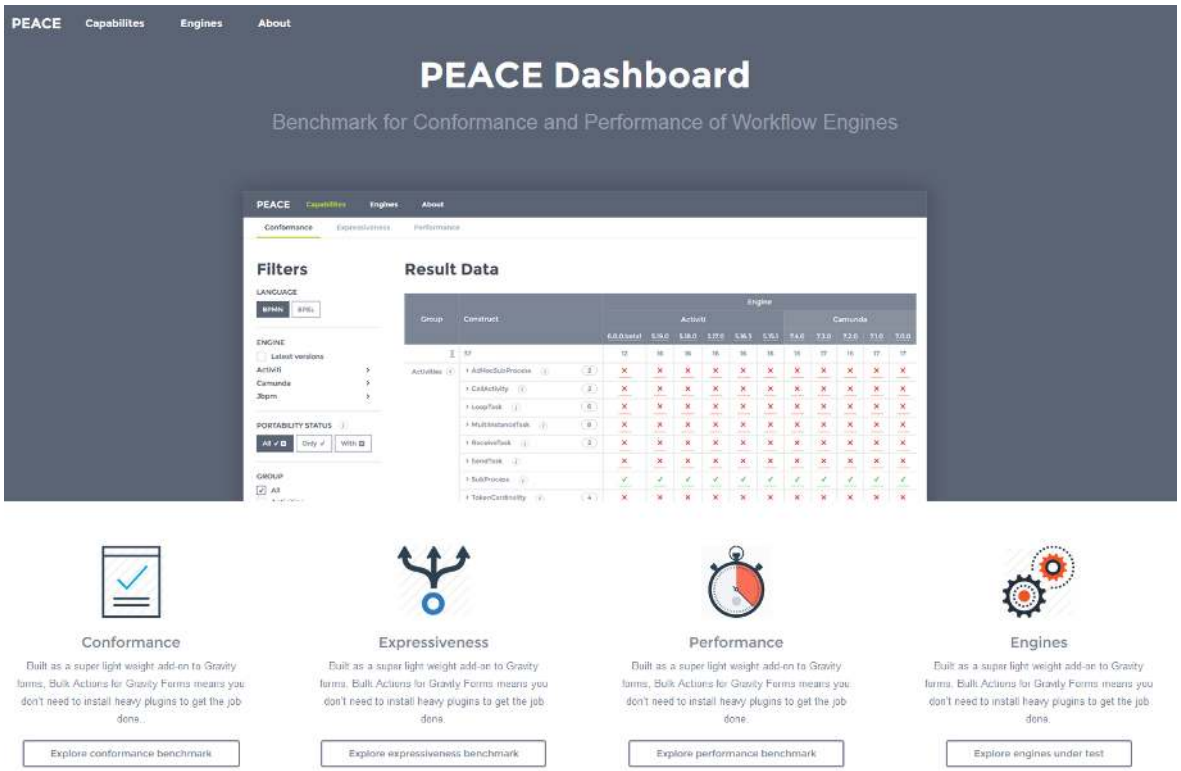


Figure 6.4.: Start page of the dashboard

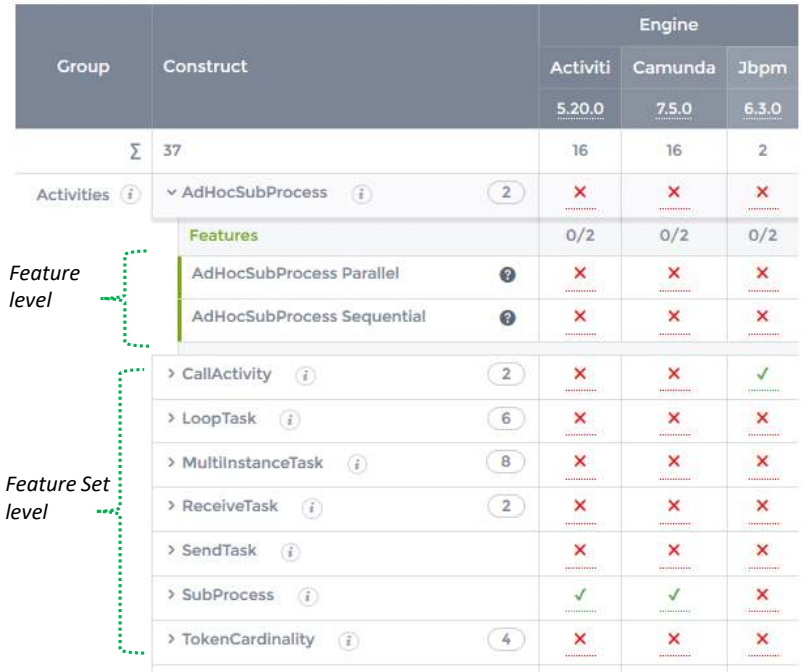


Figure 6.5.: Example of Capability Results: Conformance

with its metrics in Section 4.5.2.2). There are, however, three differences. First, a different naming is applied. Instead of using group, feature set, and feature, the expressiveness-specific terms *pattern catalog*, *pattern*, and *pattern implementation* are shown. Second, the column for the language support is

added. The data comes from the extension element *language support* of the feature and feature sets. Third, the measurement of the *support* metric is shown. The coloring is derived from the measurement of the metric *standard fulfilled*. Only if the standard is fulfilled, the rating will be colored green.

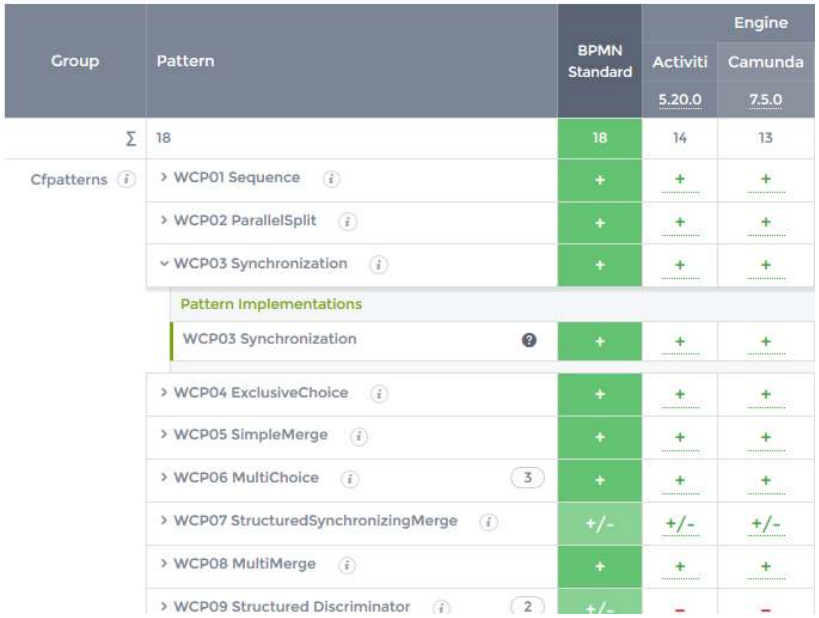


Figure 6.6.: Example of Capability Results: Expressiveness

Performance Results

The presentation of performance benchmark results is much simpler as they do not use the feature tree as already stated in Section 4.5.3.4 as part of the benchmark description. This is because aggregation of performance metrics does not make any sense. Hence, only different BenchFlow experiments (i.e., testable features; see Section 4.5.3.4) are presented in Figure 6.7. The metrics of each experiment are categorized into *performance* and *resource utilization* metrics and detailed as minima, maxima, average, standard deviation, and relative standard deviation values. This categorization is built on top of the data available in PEBL.

Drill Drown

Several methods to drill down are provided: a click on the info badge of the feature set or group shows its description, whereas the info badge on the feature shows the test, as depicted on the left in Figure 6.8. The click on a mark of a testable feature shows test results, as shown on the right in Figure 6.8. Furthermore, hovering over an engine version number shows details about that version as well.

Filters

Filters are crucial for every dashboard as they allow users to find information of interest quickly and to focus on specific parts of the result data. Therefore, each capability page provides a set of filters that allow users to explore and filter the data displayed in the table. One can filter the feature tree and the engine list for any capability. To provide a better user experience, one can search the filter options for a quicker selection. This is necessary because the conformance benchmark described in Section 4.5.2.1 already has over a

Experiment		Camunda 7.4.0			
› Micro-Benchmark					
	Performance	min	max	avg±ci	sd
	Process Duration	0	434	1.22 ± 0.02	4.21
	Throughput		1061.27		
	Number of Process Instances	--	575210	--	--
	Resource utilisation	min	max	avg±ci	sd
	CPU	24.48	56.35	48.52 ± 0.28	5
	RAM	677.1	1073.15	991.86 ± 4.24	75.21
	IO	--	--	--	--
	Size of Stored Data	--	--	--	--

Figure 6.7.: Performance benchmarking results

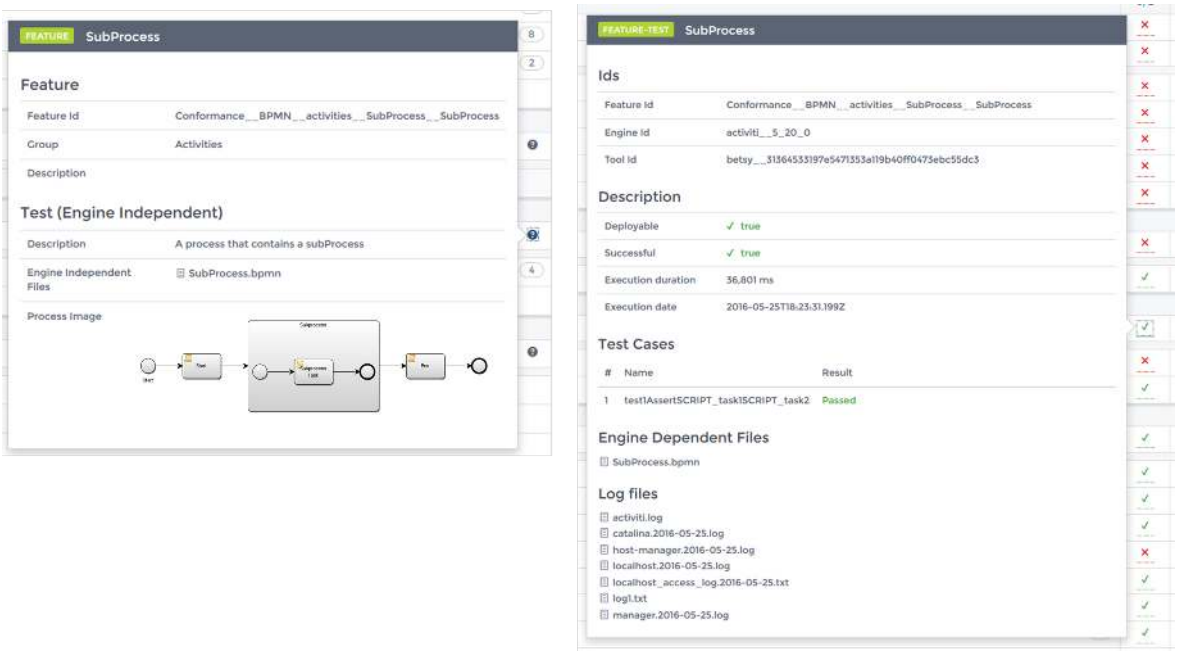


Figure 6.8.: Test (left) and Test Results (right)

hundred features. Figure 6.9 shows such filters for the conformance result page. The user normally wants to know something about the latest engine and not some older version. Consequently, all data is shown for the latest engines by default as a starting point. That selection can, of course, be modified by the user. Also, there are two custom filters: the *portability status* filter for the conformance capability (as shown in Figure 6.9) and the *standard fulfillment* status filter for the expressiveness capability. The portability status filter only allows showing the features that are *fully portable* (i.e., supported by all selected engines), *partially portable* (i.e., supported by at least one of the selected engines), and *not portable* (i.e., the support differs between the

selected engines). The standard fulfillment status works similar, but instead of focusing on the portability, it focuses on whether the pattern and pattern implementation fulfilled the language support of the standard. One can see only the pattern that *fulfilled the standard*, or the ones that *deviated from the standard*, or the ones that simply *differ* between the engines.

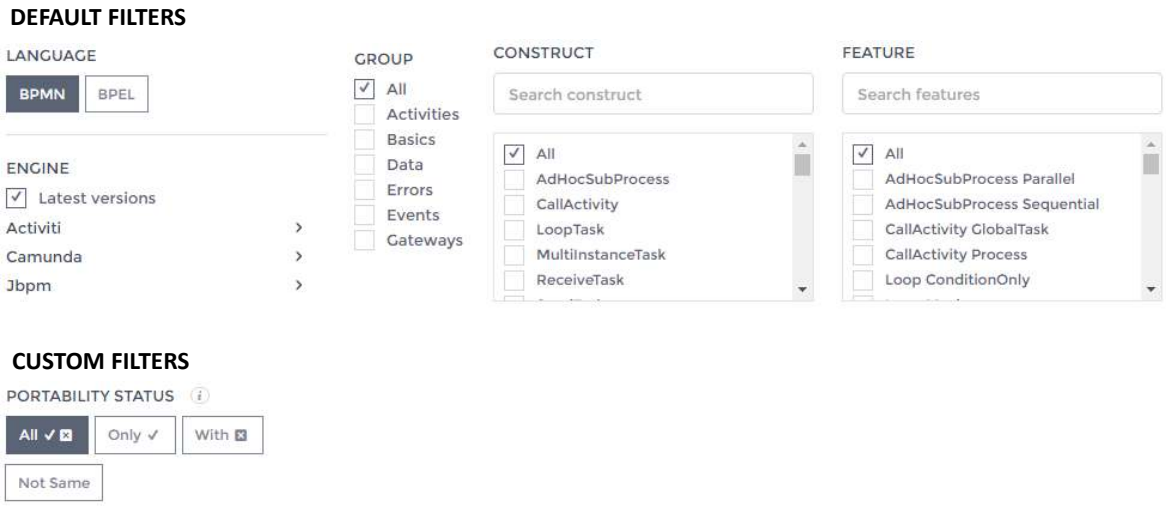
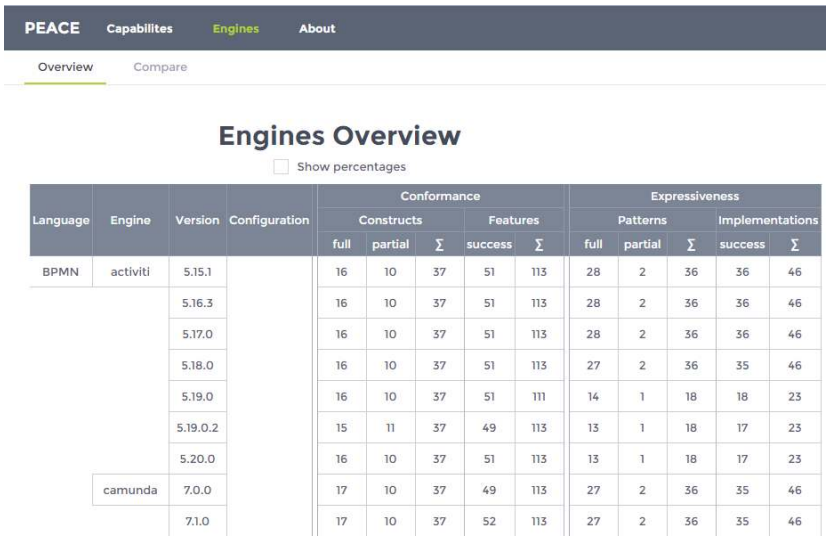


Figure 6.9.: Filters: Default and Custom Ones

The engine overview page is depicted in Figure 6.10. It comprises a table in which all versions of all engines are listed along with their most important metrics. For instance, regarding the conformance capability, it can be seen how many of the features are supported and how many features have been tested. This can be changed to a percentage-based view via the checkbox *Show percentages*.

Engine Overview



Language	Engine	Version	Configuration	Conformance					Expressiveness				
				Constructs			Features		Patterns			Implementations	
				full	partial	Σ	success	Σ	full	partial	Σ	success	Σ
BPMN	activiti	5.15.1		16	10	37	51	113	28	2	36	36	46
		5.16.3		16	10	37	51	113	28	2	36	36	46
		5.17.0		16	10	37	51	113	28	2	36	36	46
		5.18.0		16	10	37	51	113	27	2	36	35	46
		5.19.0		16	10	37	51	111	14	1	18	18	23
		5.19.0.2		15	11	37	49	113	13	1	18	17	23
		5.20.0		16	10	37	51	113	13	1	18	17	23
	camunda	7.0.0		17	10	37	49	113	27	2	36	35	46
		7.1.0		17	10	37	52	113	27	2	36	35	46

Figure 6.10.: Engine Overview Page of the PEBDASH Prototype

On the engine comparison page, two engines can be compared side-by-side as shown in Figure 6.11. In this example screenshot, *camunda BPM 7.0.0* is

Engine Comparison

compared with *camunda BPM 7.1.0*. The *Only show differences* checkbox is enabled, meaning that only the differences are shown while the commonalities are hidden. In that case, the version, ID, release date, and two signal events differ. If those are the only differences, an upgrade would be advisable, as the newer version supports two signal event types more than the previous one.

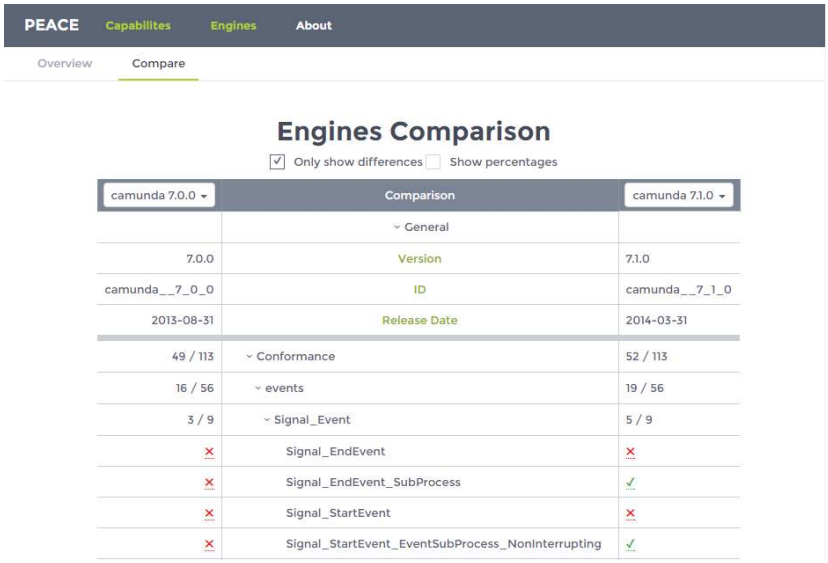


Figure 6.11.: Engine Comparison Page of the PEBDASH Prototype

Technology

The web interface of the dashboard is implemented using HTML5, CSS, and JavaScript. It is built upon the bootstrap[79] framework. The CSS definitions are managed using Sass[80]. The dashboard is made interactive through the use of JavaScript, especially relying on the *jQuery*[81] for Document Object Model (DOM) manipulation, *Underscore.js*[82] for having a richer standard library, *Moment.js*[83] for handling dates, and *Handlebars*[84] as a template system. Moreover, the implementation is modular and open for the integration of new capabilities thanks to the ability to define modules in ES6.

Architecture

The architecture of the dashboard is shown in Figure 6.12. The architecture is based on the Model-View-ViewModel (MVVM) pattern[85]. Each package within the PEBDASH corresponds to an ECMAScript 6 module, and each module resides in a separate file. The *model* fetches the JSON representation of its PEBL based database and converts it into its own representation. The filter allows selecting parts of the model based on user input. The UI consists of a tree of *components* with their *templates* using *Handlebars* and *viewmodels*. The *viewmodels* use the *filters* to select the appropriate data from the *models*

[79] `http://getbootstrap.com/`, visited 2017-3-31
[80] `http://sass-lang.com/`, visited 2017-3-31
[81] `https://jquery.com/`, visited 2017-3-31
[82] `http://underscorejs.org/`, visited 2017-3-31
[83] `http://momentjs.com/`, visited 2017-3-31
[84] `http://handlebarsjs.com/`, visited 2017-3-31
[85] `https://msdn.microsoft.com/en-us/library/hh848246.aspx`, visited 2017-3-31

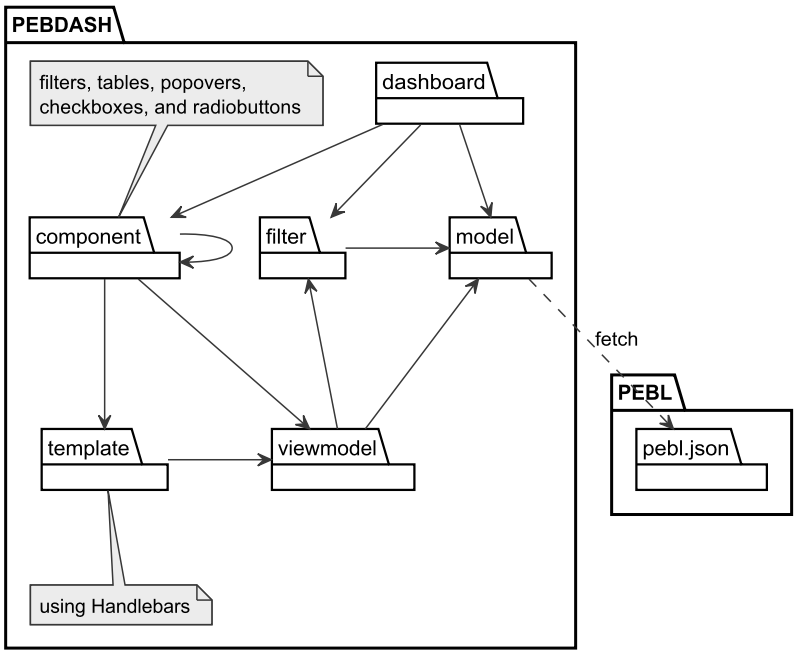


Figure 6.12.: The Architecture of the Dashboard

into a form which is easily displayable through components. A typical tree of components starts with the capability table which has a) filters that make use of checkboxes and radio buttons, b) the different popovers to provide more detail for groups, feature sets, features, tests and test results, and c) the table itself.

Limitations

The dashboard comes with a few limitations. First, although it is built for extension, a new capability will not be displayed automatically. This still requires (minor) manual integration. Second, because the dashboard has no server-side backend, it loads the whole database from the server once and processes it on the client. This can lead to performance issues[86] the more benchmarks and their results are in the database. Third, the information is shown in tables only. It does not provide any charts that could visualize the results more clearly.

6.6. Evaluation

The prototype is evaluated by checking whether all the requirements that are specified in Section 6.3 are met. These requirements are targeted at the dashboard, not the loader. However, because the loader is a necessary preprocessing step that the dashboard depends upon, it is transitively evaluated as well. In the following, all requirements are checked against the database and the dashboard prototype to determine whether they are fulfilled.

[86] See `http://josh.zeigler.us/technology/web-development/how-big-is-too-big-for-json/` for an overview how large JSON files can impact the browser performance, visited 2017-3-31.

Database: Functional & Nonfunctional Requirements

All FRs from Section 6.3 can be fulfilled with the data from the database. The results can be compared a) along the feature tree using either the feature level or any other aggregation level, b) along the engine dimension using either the latest engine, multiple versions of the same engine or any other combination of engines, and c) by drilling down to the test results and the actual engine logs. *NFR.2* is partly fulfilled through the visualization of the process models, as it makes the dashboard more visually appealing as well.

Dashboard Functional Requirements

With the engine overview page, *FR.User.1* is fulfilled as it shows all aggregated information about the engines in a highly condensed way, providing an overview of the quality of these different engines at a single glance. *FR.User.2*, *FR.User.3*, and *FR.Vendor.1* are fulfilled through the large tables available on the capability pages. These tables show similarities and differences between the latest engines on an aggregated and a more detailed level. Using the filters (see Figure 6.9) on the left side of the capability pages, different versions of a single engine can be selected. Also, one can compare two engines side by side as well, providing a view of all capabilities on a single page, with the ability to show only the differences. This functionality fulfills *FR.User.4* and *FR.Vendor.2*, giving the user the possibility to check whether to upgrade or not to upgrade and the vendor to track the progress of his product. From the information that is directly displayed on the capability page, one can drill down to the actual cause why this feature is supported or not supported. It is even possible to inspect the logs of the underlying test that was used to determine feature support. Both can be achieved through the test result page (see Figure 6.8 on the right), which lists the logs and, in case of failures, the causes per test case. Hence, *FR.User.5* and *FR.Vendor.3* are fulfilled. The engine side-by-side comparison page can also help in determining migration costs, as it allows revealing only the differences between two engines, fulfilling *FR.User.6*.

Dashboard Nonfunctional Requirements

NFR.1 is fulfilled because there is no server side component necessary to host and provide the dashboard. All data can be preprocessed through the loader, but the dashboard only relies on a standard web server that serves static files, including the JSON serialization of the PEBL database. *NFR.2* is also fulfilled because the information can be found quickly in the tables of the capability pages, which can be filtered as well. Moreover, the provided information on these pages is shown using visualizations such as images of the process models, the use of icons, or through tables for structured information.

Threats to Validity

The evaluation itself has been conducted based on the collected requirements. Hence, the evaluation depends on how well the requirements are fulfilled. This works well for highly specific requirements, but not so well for unspecified requirements such as ease of use. Hence, this evaluation should only be seen as a preliminary one that simply shows that the checklist of requirements is met, but it is no scientifically rigorous evaluation. Also, the number of experts available for coming up with the requirements is limited. Nevertheless, they

have been sufficient to define a set of requirements for such a preliminary evaluation.

The PEBDASH with its dashboard and loader enables more accessible and verifiable benchmarks. More accessible, because the results are easier to be compared with each other within the dashboard than within the PEBL representation. And more verifiable, as issues within the results can be easily checked in the comparison tables of the dashboard. This enables to perform At Least One Success (P21) with lesser effort. Hence, PEBDASH enables good benchmarks. Good Benchmark Criteria

6.7. Summary

In this section, an interactive dashboard that uses PEBL models to visualize the benchmarks and their results and the loader that loads new data into the database of the dashboard has been presented. The dashboard has been designed so that end users, developers, or researchers can compare process engines. The implementation of the dashboard was driven by requirements in the form of user stories. It has been shown that the dashboard fulfills those requirements. Hence, hypothesis H4.4 *("An interactive dashboard is a suitable form to present benchmarking results and support selection decisions.")* is supported. Conclusion

For situations to decide whether to adopt a specific engine, the dashboard alone can suffice. If the selection of the engine requires a more sophisticated decision making process including a large variety of different factors, e.g., using AHP, which is detailed in Section 2.1.3, the available data can be further processed. All data is available using the serialization of PEBL and aggregated results are computed. Hence, it can enable more complex decision making methods as well. Decision Making

Future work is subdivided into technical and organizational improvements. Regarding technical ones, it is aimed to add charts and graphs to the dashboard so that the existing data is presented in other ways. Further, it is planned to generalize the dashboard so that any new capability can automatically shown to the user without any additional effort. Moreover, it is aimed to incorporate results of other process engine benchmarks, e.g., the one created by Delgado et al. [45]. Regarding organizational improvements, the plan is to build up a community for process engine benchmarking comprising end users, developers, and engine vendors alike. This community can then drive the improvement of the dashboard, betsy, and BenchFlow through their feedback. Future Work

Efficiency is doing things right; effectiveness is doing the right things.

Peter Drucker

7. Efficient Process Engine Benchmark Framework

Parts of this chapter have been taken from [100].

In this chapter, hypothesis H4.5 *("By leveraging virtualization it is possible to improve the efficiency of a benchmarking framework significantly.")* is supported.

The Efficient Process Engine Benchmark Framework (ePEBWORK) is presented in this chapter. It is an improved and more efficient version of the Process Engine Benchmark Framework (PEBWORK) from Chapter 5. Hence, ePEBWORK builds upon the Process Engine Benchmark Language (PEBL) from Chapter 4 and the Process Engine Abstraction Layer (PEAL) from Chapter 3 as well.

7.1. Motivation

Test Isolation and Automation through Reinstallation (P9)

As shown in Chapter 5, process engine benchmarking can be performed effectively through the prototype betsy that implements an automated version of PEBWORK. The benchmarking ensures test isolation through Reinstallation (P9), meaning that fresh process engines (i.e., freshly installed and started process engines) are used for each test. Test isolation is crucial to prevent side effects (e.g., that one test case with an infinite loop cannot lead to test failures for any subsequent tests [94]). Reinstallation (P9), however, depends on a clean uninstallation procedure so that the next installation is a fresh installation. Moreover, Reinstallation (P9) solely works for engines that can be installed automatically. There are process engines, however, that are hard to install automatically, especially commercial products. The Oracle Business Process Engine, for example, is part of the Oracle SOA Suite 11gR1 middleware. The guide [193] that describes the installation of the Oracle SOA Suite requires the user to download five files summing up to five Giga Byte (GB) in total [193, p. 5–6] and to follow the necessary installation steps described on 48 pages. Hence, benchmarking such process engines is not feasible with Reinstallation (P9) because of its lack of efficiency. Another downside of Reinstallation (P9) is its effect on the time to results. Because an engine is installed, started, and terminated for each test, the time to install, time to start, and time to terminate is

multiplied by the number of executed tests, heavily influencing the overall time to result. Those timings are not neglectable. Lenhard et al. [150] showed that the installation time ranges from approximately three seconds up to more than two minutes for open source BPEL engines. This is also shown in the evaluation of betsy later in Section 7.5. The issue of the startup time is not that pressing as the issue of installation, but should not be neglected. Although the open source engines start in at most one minute as shown in Section 7.5.2, commercial ones within large middleware suites may have much longer startup times. The Oracle middleware suite requires more than ten minutes until it is up and running[87]. Because the installation, startup, and even shutdown times increase the time to result, their influence is significant. Five open source BPEL engines were evaluated regarding standard conformance with approximately 130 test cases, and this evaluation took 10 hours to complete [95]. The reason why process engines have so long and complex installation and startup procedures is that they are inherently complex middleware software products [36]. In summary, two problems have been revealed: 1) Reinstallation (P9) is not scalable to achieve test isolation, and 2) Reinstallation (P9) prohibits the benchmarking of process engines that are hard to install automatically (i.e., this holds for mostly proprietary engines such as the ones provided by Oracle and IBM) .

Efficiency through Virtual Machines (P10)

The goal is to increase execution efficiency (i.e., reduce time to results) while retaining technical effectiveness (i.e., keep the quality of results) of PEBWORK. Both, test isolation and test automation are crucial for achieving this goal because the first attributes to the *quality* of the results and the latter to the *time* to the results. Instead of achieving test isolation and automation through automated Reinstallation (P9), Virtual Machines (P10) are leveraged. Consequently, instead of *engine-dependent* and high installation, startup, and shutdown times, the availability of virtual machine snapshots that contain a fresh and running process engine allows *engine-independent* and constantly low installation, startup, and shutdown times through snapshot restoration and virtual machine termination. This approach guarantees 1) test isolation because each test runs within a freshly restored instance of a VM and 2) test automation because the snapshots and virtual machines can be either automatically created on-demand through scripts or manually upfront in case the process engine has hard to automate installation procedures, ensuring fully automated benchmarking either way. The improved version of PEBWORK is called Efficient Process Engine Benchmark Framework (ePEBWORK) which comes with its extended version of betsy called virtualization-enabled betsy (vbetsy). The improved ePEBWORK is evaluated by comparing the time to results in between betsy and vbetsy. Any time savings are compared to the overhead caused by the usage of VMs. Hence, the posed hypothesis H4.5 *("By*

[87] Measured by starting the *AdminServer* in the prebuilt Oracle VM from `http://www.oracle.com/technetwork/middleware/soasuite/learnmore/vmsoa-172279.html`, visited 2017-3-31.

leveraging virtualization it is possible to improve the efficiency of a benchmarking framework significantly.") is evaluated in this chapter.

Chapter Structure

The remainder of this chapter is structured as follows. First, Section 7.2 presents related work followed by the approach on how to tackle the stated issues using virtual machines and snapshots in Section 7.3. In Section 7.4, the details on the prototype vbetsy, the extension of betsy with virtualization techniques and DevOps [77] methods, is provided, followed by an evaluation of the performance improvements in Section 7.5. This chapter closes with a summary and future work in Section 7.6.

7.2. Related Work

Provisioning of VMs and Appliances

In the area of provisioning applications within VMs, there are two related approaches available. The most popular approach is based on script-oriented solutions, e.g., Puppet[88], chef[89], and sprinkle[90]. These have in common that the steps required to install and verify the correctness of the installation of a specific component are specified in a module. These modules may then be reused and reconfigured for more complex environmental infrastructure. Lately, however, a more service-oriented solution has emerged based on the OASIS standard TOSCA [184]. Instead of implementing these steps in scripts, the steps are implemented as BPEL processes [21]. This has the advantage that visualizing the provisioning process and monitoring its execution is natively available. For TOSCA, there exists an implementation of a modeling tool [139] and runtime [20]. The tool vbetsy leverages sprinkle for the provisioning scripts. In the future, those scripts could be converted to TOSCA processes.

Testing with VMs

Since the emergence of virtualization technology, many approaches have adopted the use of VMs for enabling or speeding up testing. Especially in the area of testing the design of a website in different browsers, it is widely applied[91]. In this case, the system under test is the web application, and the VMs are different test cases while this chapter uses VMs the other way round. VMs are used in the area of testing middleware as well. For evaluating the performance of ESBs, the corresponding ESBs are installed on the IaaS solution Amazon EC2[92] where the test itself is executed as well. Although they use VMs, they do not control them during test execution but just leverage the cheap availability of infrastructure in the cloud instead of buying physical hardware. In the context of Timeseries Databases, TSDBBench[93] uses Vagrant

[88] `http://puppetlabs.com/`, visited 2017-3-31
[89] `http://www.getchef.com/`, visited 2017-3-31
[90] `http://rubygems.org/gems/sprinkle`, visited 2017-3-31
[91] E.g., `http://browsershots.org/` or `https://www.browseemall.com/`, visited 2017-3-31
[92] `http://esbperformance.org/#ESBPerformanceTesting-Round7-PerformanceTestEnvironmentandResources`, visited 2017-3-31
[93] `https://github.com/TSDBBench`, visited 2017-3-31

to setup VMs. They rely on an extended version of Yahoo! Cloud Serving Benchmark (YCSB) [37]. Hence, that approach does not apply to the setting of this work. Another approach which controls VMs and uses snapshot restoration during test is the ruby project Virtual Machine Test Harness (VMTH)[94]. Its purpose is the automatic unit testing of provisioning scripts for Kernel-based VMs (KVMs) running solely on Linux. Although it also uses VMs and snapshot restoration in a similar way as vbetsy does, its system under test is a provisioning script which alters the VM itself while vbetsy tests process engines. Due to its focus on its specific domain, VMTH is not reusable for the problem of this work.

Testing with Containers

A more recent virtualization technology is container virtualization. Such containers make use of OS-based isolation mechanisms, resulting in a lesser overhead in comparison to VMs but with a sufficient amount of isolation [60, 293]. Because of ecosystems such as that of Docker, multiple authors (e.g., Boettiger [22], Chamberlain and Schommer [31]) suggest making reproducible research using such container technology. Furthermore, Dashevskyi et al. [40] and Rahman et al. [205] have proposed to use Docker for setting up testbeds quickly and reproducible. Although the usage of containers sounds promising, Docker only added experimental support for creating RAM snapshots at the beginning of 2017. Therefore, containers are not suited as a solution for the problem at hand yet, either.

7.3. Approach

In this section, the approach to improve the efficiency of PEBWORK is detailed. First, a short description of PEBWORK is presented, followed by an outline how ePEBWORK extends and builds upon PEBWORK. Next, because ePEBWORK relies on virtualization with VMs, the states of such VMs are defined for this approach as well.

PEBWORK

The prototype betsy of PEBWORK follows the sequential control flow procedure given in Figure 5.6 and can execute those tasks automatically. In this context, the focus lies on the five tasks that interact with the engine under test: *provide engine*, *deploy process*, *execute test cases*, *analyze logs*, and *disperse engine*. In the case of executing the test case, betsy leverages soapUI for BPEL and the PEAL API for BPMN engines. In the four remaining tasks, it interacts directly with the engine under test using the PEAL API on the same host. The actual usage of the API for each of these steps is described in pseudo code in Listing 5.1.

ePEBWORK

In ePEBWORK, PEBWORK is extended and modified with the goal to achieve a better execution efficiency (i.e, lesser time to benchmark results) while still retaining the same effectiveness. To fulfill this goal, the general idea is to leverage the usage of VMs that can be restored from HDD and RAM snapshots

[94] `https://github.com/gregretkowski/vmth`, visited 2017-3-31

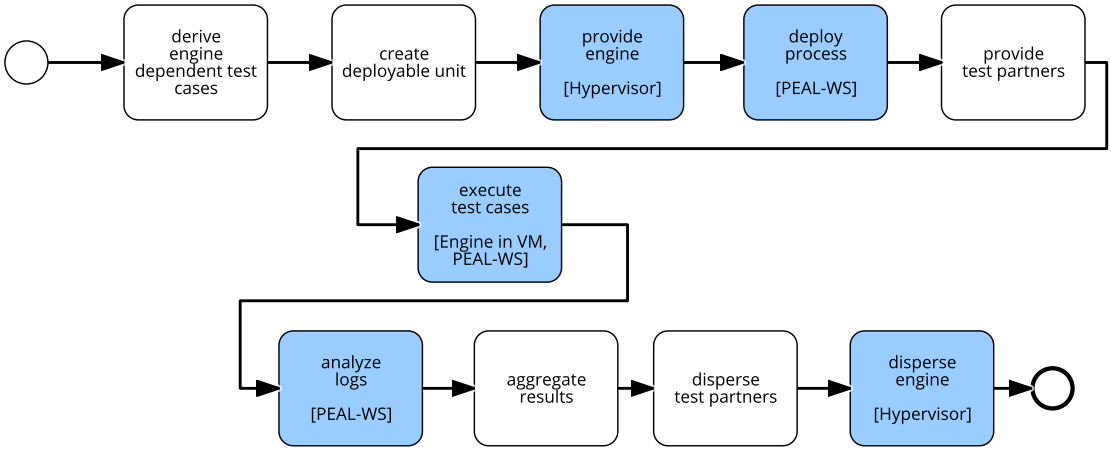


Figure 7.1.: Sequential Control-Flow Diagram of the Insider View on the Benchmarking Framework for vbetsy

in constant time instead of relying on the engine-dependent time to install, start, and stop. This idea manifests itself in the extended sequential control flow procedure shown in Figure 7.1 The modified tasks are highlighted in blue and the unchanged tasks colored in white. Normally, the tasks *provide engine*, *deploy process*, *analyze logs*, and *disperse engine* use calls to the local API of PEAL. In this case, both, *provide engine* and *disperse engine* tasks simply provide a VM with the engine already started that is fresh as well as not influenced by any previously run test and kill that VM through calls to the hypervisor instead. The situation is different for the tasks *deploy process* and *analyze logs* because the engine is running within a VM and both tasks need to interact with that engine. This is solved by providing the API of PEAL as a Web Service (WS) from within the VM so that one can initiate calls to this API from another host, reusing the same operations as in PEBWORK. The component is called PEAL Web Services (PEAL-WS). The last of the five modified steps (i.e., the step *execute test cases*) needs to know the new address of the engine as it is no longer on the same host but within the VM. The previously described tasks are also shown in Listing 7.1. The pseudo code shows how the five blue tasks are mapped to either calls to the hypervisor (for the tasks *provide engine* and *disperse engine*) or to the PEAL-WS (for the other three tasks).

Listing 7.1: Mapping of ePEBWORK Procedure Tasks to PEAL API Calls.

```
task provide engine:
    if VM has snapshot:
        call VM restore
        call VM start
    else:
        call VM start
        wait until VM is started:
            call VM snapshot

task disperse engine:
    call VM kill

```

```
13 task deploy process:
14     remote call to PEAL-WS
15
16 task execute test cases:
17     remote call to PEAL-WS [BPMN]
18     remote call to engine via soapUI [BPEL]
19
20 task analyze logs:
21     remote call to PEAL-WS
```

VM Provisioning

The VM, however, needs to be provisioned and used in the previously defined control flow. The provisioning comprises six steps: 1) creation of a VM, 2) installation of PEAL, 3) installation of the engine, 4) starting of the engine, 5) starting of PEAL-WS, and 6) creation of a HDD and RAM snapshot. It should be noted that these steps are only necessary once. If there is already a VM with the corresponding snapshot, nothing has to be done. What is more, the VMs are to be configured so that they automatically start their engine and the PEAL-WS upon startup and provide a mechanism to indicate that both are ready to receive requests. With such a mechanism in place, snapshots can be created on-demand by starting the VM and waiting for all relevant services to signal that they started. Creating snapshots on-demand is necessary as snapshots cannot be shared as easily as images of VMs because snapshots are not part of the portable OVF/OVA by the DMTF [48]. The six steps described can be fully but also partially automated. For instance, the installation of the engine could be performed manually without prohibiting fully automated benchmark execution. Hence, engines which are hard to install automatically can be covered as well.

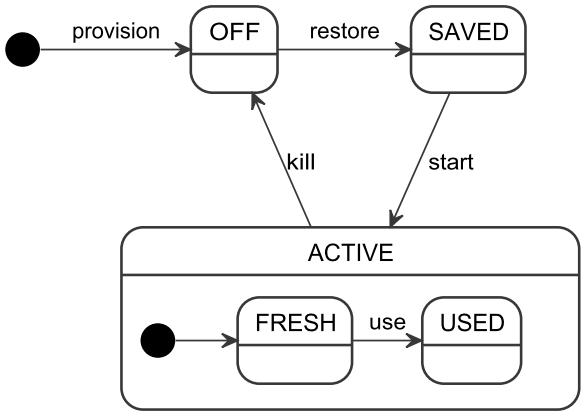


Figure 7.2.: State Machine of the VM Lifecycle of ePEBWORK

VM Lifecycle

After the provisioning, the VM is set to the *OFF* state. In that state, only the previously created snapshot can be restored, setting the VM in the *SAVED* state. In this snapshot, any software that is required for testing may already be started. Effectively, this sets up the fresh instance of the system under test: the running engine. The VM, however, is not yet usable, as it is not yet running itself. From this *SAVED* state, the VM is started, putting the VM into the *ACTIVE* state. Within this *ACTIVE* state, it is usable in a *FRESH* sub-state. In the context of this work, it is usable for exactly a single test case execution. After it has been used in even a single test case, it is not longer *FRESH*, but *USED*. When

the VM has to be renewed again, the VM must be stopped returning to its *OFF* state from where the cycle can start again. To avoid any long running stop procedures, and because a VM is an isolated and virtual environment, the easiest way to stop it is to kill the VM.

7.4. Prototype vbetsy

vbetsy

In this section, the prototype of ePEBWORK named virtualization-enabled betsy (vbetsy) is presented. That prototype vbetsy extends betsy by implementing the previously described approach that relies on VMs with snapshots. It is open source and publicly available[95].

Structure

This section is structured as follows. First, the approach to creating the VMs using methods from the DevOps movement is presented in Section 7.4.1. Section 7.4.2 presents the implementation of the provisioning and lifecycle of the VMs. Last, the limitations of vbetsy are outlined in Section 7.4.3.

7.4.1. Engine Provisioning

Provisioning

To create the required VMs for each engine under test, it is necessary to create a) a VM image, b) install the engine under test as well as the PEAL-WS, and c) export the image as a portable VM. To achieve this, methods from the DevOps movement are adopted (i.e.,the creation of the VMs including the engines is converted to code).

VM Creation

First, a minimal Linux machine, the base image, is installed within a VM as the foundation for all subsequent steps. It is granted 4 GB RAM and a single core processor[96]. Moreover, the audio capabilities are deactivated and the network is configured using a NAT adapter. This base image[97] is built upon an Ubuntu Server 12.04.2 LTS. It requires a user with sudo privileges and an SSH server.

VM Provisioning

Second, the BPEL engine and the PEAL-WS need to be provisioned on the base image. Figure 7.3 shows the dependency graph of all BPEL engines (gray background), the PEAL-WS (black background) and other software products (white background). Each edge is a *depends on* relation, e.g., Apache ODE 1.3.6 depends on Tomcat 7.0.26. This dependency graph has been converted to executable provisioning scripts that can install any node alongside its direct as well as transitive dependencies. These provisioning scripts are implemented with sprinkle v0.7.6.2. The DSL of sprinkle allows declaring such a dependency graph straightforward in its executable constructs. Each node in Figure 7.3 is

[95] `https://github.com/uniba-dsg/betsy/`, visited 2017-3-31

[96] For benchmarking performance, this, of course, would require a different configuration, but for benchmarking capabilities such as conformance, this suffices.

[97] Base image is available at `https://lspi.wiai.uni-bamberg.de/svn/betsy/ova/basevm.ova`, visited 2017-3-31.

implemented as a sprinkle package which includes the download, installation, configuration, and start of the component through a sequence of CLI calls. The edges between the nodes are directly ported to dependencies between these sprinkle packages. Based on these packages, so-called sprinkle policies have been created for each VM. If executed, those policies automatically install both, the engine and the PEAL-WS. The sprinkle scripts containing all packages and policies are open source and publicly available[98]. Next, all VMs are provisioned by applying each of the six policies on a separate base VM.

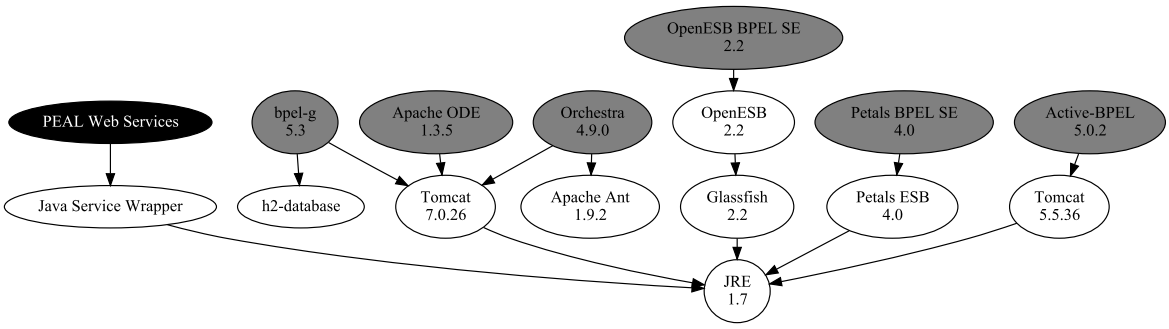


Figure 7.3.: Deployment Topology of all BPEL Engines and the PEAL-WS

On-Demand Snapshots

Third, the six provisioned VM images have been exported to portable Open Virtual Appliance (OVA) archives that are publicly available[99]. During the execution of vbetsy, these OVA[100] archives are downloaded from the given URL, imported into VirtualBox, and started. However, these host independent and therefore portable OVA packages cannot contain host dependent snapshots. To circumvent this problem, the snapshots are created on-demand by vbetsy itself. This is possible as the BPEL engines and the PEAL-WS are configured to start automatically during the startup of the VM. Thus, vbetsy downloads the VM, imports it into VirtualBox, and starts it. When the engine and the PEAL-WS are up and running, it creates a snapshot which is then reused for any subsequent tests.

Automation

To make this process repeatable, it is automated. As the base VM only has to be created once, this step can be automated by reusing the previously created base VM. The other two steps are implemented in the form of a Groovy script[101] which can automatically provision a VM for any of the six BPEL engines and export the image as an OVA archive. For engines that require manual installation steps, the archive can also be created manually. This allows integrating a commercial behemoth such as the Oracle Process Manager.

PEAL-WS

Instead of the PEAL-WS described in Section 3.4, its predecessor has been used for vbetsy. Instead of relying on WSDL-based message exchanges, this

[98] `https://github.com/uniba-dsg/betsy-engines`, visited 2017-3-31
[99] `https://lspi.wiai.uni-bamberg.de/svn/betsy/ova`, visited 2017-3-31
[100] An OVA package comprises files complying to the OVF [48, line 403].
[101] `https://github.com/uniba-dsg/betsy/tree/master/src/main/groovy/betsy/bpel/tools/VirtualMachineInstaller.groovy`, visited 2017-3-31

predecessor only supports the deployment of process models and the collection of the logs over TCP. Moreover, it builds upon Java 7 instead of Java 8. Because vbetsy only supports BPEL engines which can be tested remotely without PEAL, those two remote calls suffice.

7.4.2. Execution Model of VMs

VM Execution Model

To harvest the benefits of using VMs in contrast to the bare metal approach of betsy, it is necessary to look at the lifecycle model of a VM within vbetsy that is shown in Figure 7.2. Any VM used in vbetsy must only change according to the model provided in Figure 7.2. To guarantee this, vbetsy ships with an interface (see Figure 7.4) that corresponds to the state transitions of the described figure. This interface is implemented by vbetsy on top of VirtualBox to control any VM of VirtualBox according to the proposed execution model of VMs. The engine IDs are used to name and identify the VMs. Moreover, all the VMs are namespaced under the group vbetsy to keep them separated from other VMs within VirtualBox.

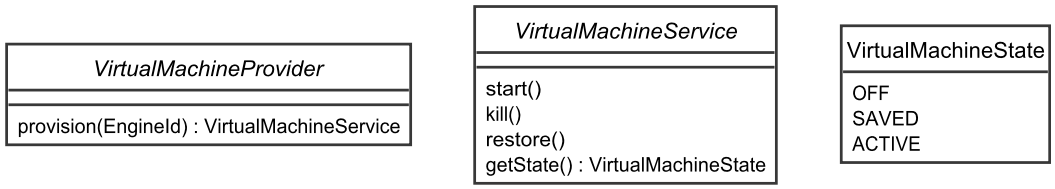


Figure 7.4.: Common Interface for the VM Execution Model

7.4.3. Limitations

Portability

The approach works with any correctly created VM image that fulfills the interface and that is bundled as a portable OVA file. However, the system that imports and uses this image must be able to fulfill the hardware requirements for the image. Thus, the portable image has a minimum hardware dependency. Moreover, the snapshots themselves are not portable and have to be created per machine. Consequently, upon updating the VM image, any snapshot based on this VM has to be recreated, causing additional overhead. Alternatively, a newer snapshot can be created on top of an existing one. This evolution strategy, however, can only be used on single machines as the snapshots are system-dependent.

IaaS

The approach is a good fit for the cloud and its available IaaS products because new instances of machines can be easily spawned and discarded. However, major IaaS vendors (e.g., Amazon with its Amazon EC2) solely support the creation of disk snapshots but do not offer RAM snapshots[102].

[102] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-creating-snapshot.html`, visited 2017-3-31

Hence, only a part of this approach is directly implementable on such IaaS systems. Also, this approach does not take security into account, which would be necessary when leveraging such IaaS offerings.

The prototype works only for these six BPEL engines. It has not been ported to the BPMN engines. This, however, is straightforward as both engines do implement the uniform PEAL API. BPEL only

7.5. Evaluation

This section presents the evaluation of the feasibility of the approach and tool by determining the effects on a) install as well as start time (i.e., the main objective), and b) test time (i.e., any side effects due to the virtualization overhead). Evaluation

This section is structured as follows. First, the evaluation method is outlined in Section 7.5.1. The results of the install, start, and stop operations are provided in Section 7.5.2, followed by the results of the deploy, test, and collect operations in Section 7.5.3. This section is concluded with the threats to validity in Section 7.5.4 and a summary in Section 7.5.5. Section Structure

7.5.1. Method

To prove that it is possible to reduce both install and start time significantly, the Aptitude Test (P8) is executed five times for the two variants: vbetsy and betsy, or in other words, with and without virtualization. This test is executed on the six BPEL engines detailed in Section 7.4. The machine, on which the experiment is conducted, has an Intel i7-2600 processor, 16 GB of RAM, and a Western Digital WD10EALX hard drive. Software-wise, it is equipped with Windows 7 64 bit, the Java JRE 7u45, soapUI 4.6.2, and VirtualBox 4.2.16 which are required by betsy and vbetsy, respectively. Also, a CI server has been setup to help orchestrating the execution of the different test runs and gathering the produced results. The experiment uses the CI server Jenkins[103] CI v1.545 along with Git for Windows[104] v1.8.5.2 for version control. Method

7.5.2. Results for Install, Start, and Stop

The results of the experiment regarding the engine lifecycle tasks install, start, and stop are shown in Table 7.1. Each row gives the durations of the three tasks and their sum for all six engines executed with or without virtualization (virtual vs. local). These values are given in seconds and are the average of three values from the five runs because both, the highest and the lowest value are discarded Results

[103] `http://jenkins-ci.org/`, visited 2017-3-31
[104] `http://msysgit.github.io/`, visited 2017-3-31

to reduce inaccuracy of measurement. These values are stable enough for this evaluation as the relative standard deviation is below 19% for 70 of the 72 values. Although the two other values have a higher relative standard deviation, they are below one second. Hence, they do not affect the overall results. The last column contains the difference between both variants in seconds, denoting the improvements made by the proposed approach. Moreover, the averages, min as well as max values, and standard deviations per task over all engines are presented as well.

Table 7.1.: Average Execution Time in Seconds of the Engine Lifecycle Tasks per Engine without/with Virtualization

	betsy				vbetsy				betsy - vbetsy			
Engine	install	start	stop	Σ	install	start	stop	Σ	install	start	stop	Σ
ActiveBPEL	14.12	5.02	0.52	19.7	0.30	1.86	0.58	2.7	13.82	3.16	-0.05	16.9
bpel-g	3.74	9.04	0.47	13.3	0.33	1.72	0.47	2.5	3.42	7.32	0.00	10.7
Ap. ODE	5.67	9.30	0.47	15.4	0.25	1.70	0.23	2.2	5.42	7.61	0.23	13.3
OpenESB	129.11	26.52	7.96	163.6	0.21	1.82	0.46	2.5	128.90	24.70	7.50	161.1
Orchestra	18.06	12.37	0.49	30.9	0.22	1.93	0.23	2.4	17.83	10.44	0.26	28.5
Petals ESB	7.05	19.43	0.48	27.0	0.25	1.82	0.28	2.3	6.81	17.61	0.20	24.6
average	29.63	13.61	1.73	45.0	0.26	1.81	0.37	2.4	29.37	11.80	1.36	42.5
min	3.74	5.02	0.47	13.3	0.21	1.70	0.23	2.2	3.42	3.32	0.24	11.1
max	129.11	26.52	7.96	163.6	0.33	1.93	0.58	2.7	128.90	24.59	7.38	160.9
std. dev.	49.04	7.94	3.05	58.5	0.04	0.09	0.15	0.2				

betsy

Before using virtualization techniques, betsy relied only on local installation, startup, and shutdown procedures. The six engines can be put into three groups according to their total lifecycle time. bpel-g (13.25s), Apache ODE (15.43s), and ActiveBPEL (19.66) lead the field with at most 20 seconds, followed by both, Petals ESB (26.96s) and Orchestra (30.91s) with approximately thirty seconds. OpenESB (163.59s) comes in last as it requires almost three minutes on the test machine. These numbers reflect the complexity of the runtime container of the engines, as the ones of the top group, namely Apache ODE, bpel-g, and ActiveBPEL, run on a lightweight servlet container, whereas the engine that came in last, the BPEL engine of the OpenESB, is running within an ESB that is deployed onto a heavy-weight application server. A lightweight container, however, does not guarantee fast install, start, and stop times as Orchestra shows, although a BPEL engine within an ESB does not necessarily lead to long install, start, and stop times, as shown by Petals ESB. What is more, the major impact factor of the total time varies from engine to engine. For OpenESB, Orchestra, and ActiveBPEL, the install time is the driving factor. In contrast, the start time is higher than the install and stop time for the other three engines. The time to stop an engine can almost be neglected as it is .5 seconds for all engines, except for OpenESB which requires approximately eight seconds to shutdown. This is because OpenESB is stopped gracefully via a shutdown method while the OS processes of the other engines are simply

killed. In the min, max, and standard deviation values, the wide range of these engine-dependent durations can be observed. The install time has a range of approximately 125 seconds, followed by the range of the start task which accounts to approximately 25 seconds. The stop task has the lowest range of approximately seven seconds. For the install and the stop task, the standard deviation is higher than the average, showing that there are large differences in the base values, although the start task is balanced around 30 seconds.

The picture changes when looking at the numbers measured with vbetsy. vbetsy The install time ranges between .21s and .33s, the start time between 1.70s and 1.93s, and the stop time between .23s and .58s. All engines now have almost the same duration regardless of their previous durations. Instead of having wide ranges, the durations can be seen as constants. The engine-dependent durations have been converted to engine-independent values. The highest gain is achieved for OpenESB which saves 161.10 seconds. The other engines saved between approximately 10s up to 28s.

7.5.3. Results for Deploy, Test, Collect

Next, the possible side effects of this approach are investigated by looking betsy at the other engine-related steps of the betsy testing process, namely, at the deploy, test, and collect step. In Table 7.2, the timings of the test step are shown using the same columns as in the previous table. Orchestra, OpenESB, and Apache ODE have the fastest deployment process which takes at most four seconds. Petals and bpel-g form the group in the middle with 7.05s and 9.52s whereas ActiveBPEL comes in last with 18.16s. The durations of the test task do not vary as much as the deploy task, ranging only from .77s to 2.32s. The collect task is executed fast as it copies files from one folder to another on a single hard drive.

Table 7.2.: Average execution time in seconds of the engine actions and the test task per engine before and after using virtualization

	betsy				vbetsy				betsy - vbetsy			
Engine	deploy	test	collect	Σ	deploy	test	collect	Σ	deploy	test	collect	Σ
ActiveBPEL	18.16	0.79	0.02	18.97	20.18	0.83	0.19	21.20	-2.03	-0.04	-0.16	-2.23
bpel-g	9.52	0.86	0.02	10.40	9.01	0.93	0.18	10.12	0.50	-0.07	-0.16	0.27
Apache ODE	3.53	1.52	0.05	5.10	4.01	1.92	0.13	6.06	-0.48	-0.39	-0.09	-0.96
OpenESB	4.04	0.84	0.06	4.93	7.97	0.85	0.17	8.99	-3.93	-0.02	-0.11	-4.05
Orchestra	2.53	0.77	0.02	3.33	7.30	0.96	0.20	8.46	-4.76	-0.19	-0.17	-5.13
Petals ESB	7.05	2.32	0.03	9.41	9.66	3.38	0.12	13.16	-2.61	-1.06	-0.08	-3.75
average	7.47	1.18	0.04	8.69	9.69	1.48	0.16	11.33	-2.22	-0.30	-0.13	-2.64
min	2.53	0.77	0.02	3.33	4.01	0.83	0.12	6.06	-4.76	-0.06	-0.09	-2.73
max	18.16	2.32	0.06	18.97	20.18	3.38	0.20	21.20	0.50	-1.06	-0.14	-2.23
std. dev.	5.84	0.63	0.02	5.74	5.51	1.02	0.03	5.36				

vbetsy Investigating the right part of Table 7.2, a clear overhead is visible. All durations, except for a single value[105], have increased. The collect task takes .13 seconds longer on average and the test task .3 seconds. The largest effect of the extension of betsy on the duration of these tasks is seen for the deploy task as deployment takes 2.22 seconds more to complete. For the collect task, the changes are completely neglectable whereas both the deploy and test task have increased up to almost five seconds in a single case.

7.5.4. Threats to Validity

BPEL only There are threats to validity regarding the evaluation. First, the evaluation and the prototype vbetsy only include BPEL engines, leaving BPMN engines at the side. This, however, is not an issue. Because of PEAL, it is straightforward to interact with either of them through the same API. The outcome of the evaluation should not change, and the benefits for the BPEL engines should be achievable for the BPMN engines, too, because the BPMN engines have similar install, start, and stop times as their BPEL counterparts.

Different OS Second, the evaluation has been conducted on a Windows host and Linux-based VMs. Because of this, the engines were running on Windows with betsy and Linux with vbetsy. This could affect the evaluation results and accounts for the outlier in which case the startup procedure was faster on Windows than on Linux for bpel-g. The results, however, were so significant that the differences of speed in file system operations or startup of Java applications does not matter that much in between Windows and Linux.

Efficiency Third, the execution efficiency has only been evaluated by performing the Aptitude Test (P8) that is the simplest test. Using more complicated tests would not have made any difference regarding the engine-specific steps such as *provide engine* and *disperse engine*, only the overhead of the actual test execution could be a little bit higher if there are more interactions. Nevertheless, the overall significant benefit should not be influenced by this.

Effective-ness With the efficiency of the proposed approach been shown, the question is whether the *effectiveness* is retained. The difference between betsy and vbetsy is whether the calls are made locally through PEAL vs. remotely through PEAL-WS and directly with the engine vs. through the hypervisor. Because of code reuse, the same code is executed but initiated at different points in time (VM provisioning and snapshot creation) and triggered from different points in space. Hence, although no experiment has been conducted, the argument is sufficient to state that the effectiveness is retained.

[105] Investigation showed that bpel-g performs faster on the Ubuntu VM as on the Windows host.

7.5.5. Discussion

As vbetsy, in contrast to betsy, decreases execution time at the expense of space, this approach requires more RAM and disk space. Although additional disk space and RAM for using VirtualBox are neglectable, the VMs themselves introduce a major overhead due to the OS in the VM onto which the engines are installed. As RAM is more scarce than disk space, RAM may become a bottleneck, especially when testing multiple engines at the same time. RAM Usage

Replacing the HDDs used in the evaluation with SSDs will reduce the durations for betsy and vbetsy, as both rely heavily on IO throughput. However, it is unknown whether it increases or decreases the advantages of vbetsy over betsy. The additional overhead of the OS of the VM is also influencing the latency for any communication with the engine under test. Thus, the latency for the actual test step is increased. Because this is inherent to the proposed approach, this has to be taken as a given. SSD vs. HDD

The ePEBWORK with its vbetsy prototype enables more affordable, scalable, and repeatable benchmarks. More affordable because the costs to run a benchmark are reduced time-wise. This makes the benchmark more affordable in situations in which time to result is important such as CI environments that build and determine the quality of an engine set up by the engine vendor to ensure that quality is not degraded while new features are being added. And more scalable as engines with manual installation steps can be included as well. Hence, commercial products with complex installation routines can take part in the benchmark as well. And last, more repeatable because the level of test isolation is even higher with Virtual Machines (P10) than just doing Reinstallation (P9) on the local FS. Hence, ePEBWORK enables good benchmarks. Good Benchmark

Table 7.3.: Overall Reduction in Test Case Execution Time

	Active BPEL 5.0.2	**bpel-g 5.3**	**Apache ODE 1.3.5**	**OpenESB 2.2**	**Orchestra 4.9.0**	**Petals ESB 4.0**
betsy	38.63s	23.65s	20.53s	168.53s	34.24s	36.36s
vbetsy	23.94s	12.64s	8.23s	11.48s	10.84s	15.50s
Δ in s	14,69s	11,01s	12,30s	157,05s	23,40s	20,86s
Δ in %	38%	47%	60%	93%	68%	57%

When comparing the duration increase of the deploy, test, and collect tasks with the decrease regarding the install, start, and stop tasks, it is revealed that the overhead is neglectable. Because vbetsy still saves at least eleven and at most 157 seconds for executing a single test case compared to betsy. Looking at the percentages, vbetsy can reduce the test case execution time including the *setup*, *test execution*, and *teardown* phases dramatically as shown in Table 7.3. Improvements range between 38% for ActiveBPEL and 93% for OpenESB, Summary

whereas the other four engines have a reduction of at least 47% and at most 68%. To sum up, the results of the experiment show that it is possible to create fresh and started instances of such software in a timely fashion independent of any complex installation or startup procedure using virtualization techniques. Furthermore, the overall execution time is reduced between 38% and 93% despite the additional virtualization overhead. Speaking in averages, the time to result for a single test on a single engine drops from 53.7 seconds down to 13.8 seconds on average, resulting in a saving of 39.9 seconds on each test. With vbetsy, the evaluation in Section 5.5.2 would take only 4801 minutes or 80 hours or 3.3 days instead of 23.8 days. This optimization is especially helpful for continuous integration as the evaluation of a single BPMN engine can be performed in about an hour and a single BPEL engine within at most three hours.

7.6. Summary

Conclusion

This chapter presented an approach which provides fresh and started instances of process engines in an effective and efficient manner to support test isolation and automation. By evaluating the install, startup, and shutdown times of six BPEL engines with and without the proposed approach, it has been shown that it is possible to convert the software dependent install, startup, and shutdown times to constants. Nevertheless, there is an increase in latency due to the virtualization overhead for executing the actual test. However, this increase is neglectable as the gain during the *setup* and *teardown* phases of the test more than outweighed the loss in the *test execution* phase, leading to time savings ranging from approximately 11s (38%) up to 157s (93%) in total. In summary, this supports hypothesis H4.5 *("By leveraging virtualization it is possible to improve the efficiency of a benchmarking framework significantly.")*.

Future Work

Future work comprises four aspects: i) reducing the overhead, ii) creating a more generic approach, iii) increasing reusability of the provisioning scripts, and iv) increasing the efficiency through parallelization. Firstly, using VMs inherently reduces performance due to the additional overhead of two OSs. In the future, it is planned to reduce this overhead by moving from OS-based to container-based virtualization allowing to host multiple isolated containers on top of the host OS. However, Docker only introduced RAM snapshots as described in Section 2.4.2 at the beginning of 2017. Hence, they are not included in this work. Secondly, both, the ePEBWORK approach and tool vbetsy are extensions to PEBWORK and betsy and therefore highly tied to them. To achieve separation of concerns, it is planned to extract a more generic approach and tool that solely handles test setup and teardown with VMs. Thirdly, the provisioning of the six open source BPEL engines is encoded within sprinkle scripts and not easily reusable. In the future, the reusability will be increased by converting the sprinkle provisioning scripts to reusable TOSCA [184] artifacts

and by storing them in public TOSCA type repositories. Fourthly, in addition to leverage virtualization, one can also leverage parallelization to improve the efficiency. There is concurrency potential as shown in Figure 5.7. Such optimizations could work orthogonally to the ones presented in this chapter.

There are only patterns, patterns on top of patterns, patterns that affect other patterns. Patterns hidden by patterns. Patterns within patterns.

Chuck Palahniuk

8. Process Engine Benchmarking Pattern Candidates

Parts of this chapter have been taken from [102].

In this chapter, hypothesis H4.6 *("Patterns are a suitable form to describe the central elements of process engine benchmarking.")* is supported.

The Process Engine Benchmarking Pattern Candidates (PEBPATT) represent the collected knowledge of the Process Engine Abstraction Layer (PEAL), the Process Engine Benchmark Language (PEBL), the Process Engine Benchmark Framework (PEBWORK), and the Efficient Process Engine Benchmark Framework (ePEBWORK) in the form of patterns.

8.1. Motivation

Over the last few years, different process engine benchmarking approaches [16, 62, 95] have been developed. Typically, the authors solved these challenges in a similar or even the same way. As a result, there are proven solutions available for the challenges in this domain – a breeding ground for *patterns* [5, 75]. By effectively converting the solutions to patterns (i.e., identifying patterns), the kernel of these solutions can be captured in a shareable form so that the author of the next benchmark can build upon it, using it in his vocabulary to better communicate with other benchmark authors, or as a guideline for implementing his own benchmark or benchmarking framework.

Chapter Structure

The remainder of this chapter is structured as follows. First, an overview of patterns and their typical form of description is given in Section 8.2. In Section 8.3, the participants and challenges in process engine benchmarking are described, for which the pattern candidates in Section 8.4 are solutions to. The set of identified pattern candidates is discussed and evaluated in Section 8.5. This chapter concludes with a summary in Section 8.6.

8.2. Patterns

Pattern History

Almost forty years ago, Alexander [5] captured reoccurring solutions to problems in different kinds of buildings in the form of patterns. Twenty years later,

Gamma et al. [75] adopted that form of capturing reoccurring solutions to problems for the domain of software design via design patterns. With their seminal book [75], they started a pattern movement that is still going on in the world of software today. Within, however, the area of process engine benchmarking, patterns, pattern languages, or pattern catalogs are still missing.

Pattern Description

Over time, and because of the popularity of identifying all kinds of different patterns, pattern catalogs, and pattern languages in a plethora of domains, theories about patterns have emerged. For instance, Kohls [130, 131] digs into the structure of patterns and compares them to alternative routes on a map. Others such as Meszaros and Doble [170] have written a guide on how to write a pattern. They propose that a pattern description must contain a *name*, a *problem*, a *context*, *forces*, and a *solution*. *Examples* and *relations* are optional. In this work, the description that is used for the pattern candidates comprises a *name*, a *problem*, a *solution*, *relations*, and an *example*. That description exceeds the advice of Meszaros and Doble [170] because it provides examples and relations to other patterns. However, although the *context* and the *forces* are normally mandatory for the pattern description, they are explicitly excluded because only pattern candidates, a preliminary form of patterns, are identified. These candidates have to be converted to full patterns through collaboration with domain experts in the future.

8.3. Problems in Process Engine Benchmarking

This section reiterates the typical challenges that are faced when designing a benchmark for process engines or a framework for process engine benchmarks. These challenges act as drivers for identifying pattern candidates within the domain of process engine benchmarking.

8.3.1. Big Picture

The big picture of process engine benchmarking as shown in Figure 8.1 comprises four main elements: *tests*, the *engines* under benchmark, the *benchmarking procedure*, and the *results* of the benchmark.

The benchmarking procedure is the central element. It can be seen as an IO system with both, the tests and the engines as the inputs and the results as the output. The expected or even desired behavior of the target of the benchmark (i.e., the engines) is encoded into *tests*. In other words, the tests are the specification the engines are benchmarked against. The engines are the different alternatives that are being made comparable through the benchmark results.

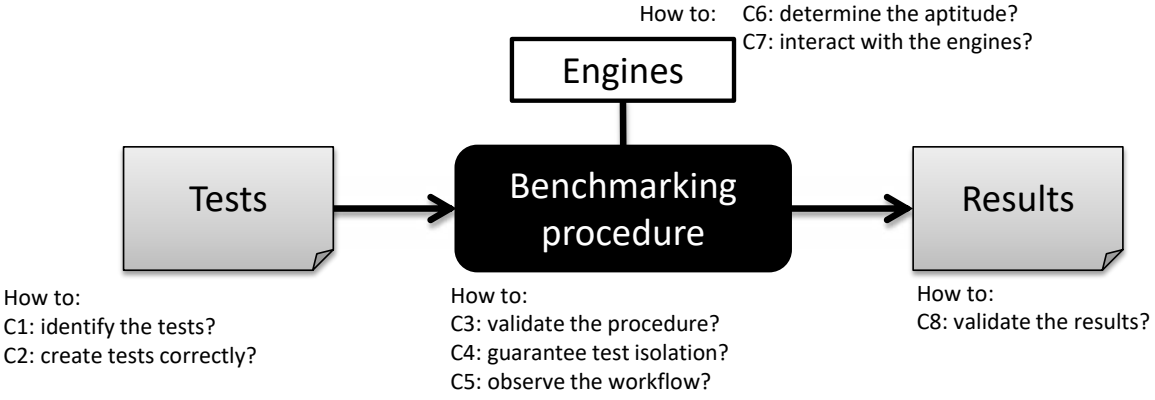


Figure 8.1.: Big Picture of Process Engine Benchmarking

8.3.2. Challenges

Each time one is building and conducting a process engine benchmark, the same nontrivial problems arise. Such problems are reoccurring challenges and the perfect starting ground for identifying reoccurring solutions (i.e., pattern). Figure 8.1 lists the eight most important challenges (C1 to C8).

Tests

In benchmarking, the tests should represent realistic usage scenarios so that the results do have value for real world decisions [7, 47, 238]. Designing such tests requires considerable effort because they should not contain any issues which may return wrong and misleading results. This is crucial because even seemingly minor issues may result in flawed data. Consequently, the two major issues are how to *identify the tests (C1)* and *create tests correctly (C2)*.

Procedure

Analogous to the tests, it is critical as well that the benchmarking procedure is not compromised through issues which cause flawed results. The procedure itself needs to make use of and apply quality assurance methods, i.e., *validate the procedure (C3)*. Also, it should *guarantee test isolation (C4)* so that the results do not depend on the execution order of the tests and that side effects of a test cannot impact the result of any other test. This includes that the tests should be independent of the execution mode being it sequential or parallel as well. Especially for large test sets, parallel execution of the tests is recommended to obtain the results timely. Last, it is important that the progress of the benchmark as a whole and that of single tests can be monitored (i.e., *observe the process (C5)*). This is required a) for continuation purposes so that, for instance, the crash of an engine during a single test should not tamper the results of other engines and the benchmark can be proceeded nonetheless, and b) for evaluation purposes so that the results can be computed accordingly.

Engine

The challenges targeting the engines are how to *determine the aptitude (C6)* of an engine and how to *interact with the engines (C7)*. For an engine to participate in a benchmark, it has to proof its aptitude. Hence, the entry level for participating in the benchmark has to be defined at that point. If an engine is suitable, it has to be interacted with during the benchmark, and

its execution has to be controlled and monitored, e.g., through exchanging messages, executing API calls, or evaluating logs. Especially for determining the test results, it is of paramount importance to check the outcome of any test assertion and to detect any failures that might have occurred. In other words, the state of the engine has to be observable.

Last, it is important to *validate the results (C8)* as well so that the decision-relevant information stemming from the raw data produced by the benchmarking do not suffer from any flaws. Such flaws can happen at the time of converting or aggregating the raw data to more decision-relevant information, or they might even reveal issues within the tests, engines, or the benchmarking procedure itself. Results

8.4. Process Engine Benchmarking Pattern Candidates Catalog

This section contains the description of 21 Process Engine Benchmarking Pattern Candidates. These pattern candidates are solutions to the challenges from the previous section and grouped according to the four elements of a benchmark from Section 8.3.1 into test, benchmarking procedure, engine, and results pattern candidates. The *test* pattern candidates in Section 8.4.1 comprise solutions to identify and assure the quality of test cases for a benchmark, whereas the *benchmarking procedure* pattern candidates are concerned about automatically conducting the benchmark in Section 8.4.2. Ways to incorporate the different engines into the benchmark are described with the *engine* pattern candidates in Section 8.4.3, and the *results* pattern candidates comprise solutions for the result data validation in Section 8.4.4.

Each pattern candidate is identified through its *name*. The challenges (see Section 8.3.2) it provides a solution to are listed as the *problems* the pattern candidates target. Moreover, the *solution* contains the steps to solve the problem in an abstract form. Concrete *example*s of the abstract solution are given as well. Last, when appropriate, the *relations* between patterns are described, too. Pattern Candidate Description

The extracted pattern candidates have been peer reviewed by researchers from two other universities in Europe (i.e., Karlstadt University and University of Stuttgart) and as part of the publication of the conference paper [102]. It is planned for future work to evaluate the pattern candidates more thoroughly by discussing them with additional domain experts and through the publication at pattern conferences such as EuroPLoP [103]. Pattern Quality

8.4.1. Tests Pattern

To *identify the tests (C1)*, one can determine the constructs of a language and apply *Configuration Permutation (P1)* or use *Reoccurring Constructs (P2)*

to identify the most used constructs and their configuration. To *create tests correctly (C2)*, one can derive tests using *Stub Extension (P3)* or *Mutated Existing Test (P4)*, which both ensure a certain degree of correctness. Applying *Open Sourcing (P5)*, *Expert Review (P6)*, and *Automatic Static Analysis (P7)* strengthen the degree of correctness further. What is more, the latter patterns can be applied independently of each other.

Pattern Configuration Permutation (P1)

Problem Identify the tests (C1)

Solution Identify a construct. Permutate all configurations of a construct. Each permutation is a test.

Example This pattern was applied in betsy for BPEL and BPMN standard conformance tests. For instance, in BPMN, there is the construct *exclusive gateway* which can be configured in three ways resulting in three tests: (i) *standard* with all outgoing sequence flows having conditions, (ii) 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.

Pattern Reoccurring Constructs (P2)

Problem Identify the tests (C1)

Solution Gather a large corpus of processes. Identify the reoccurring elements in these processes. Tests are created based on the most important (i.e., reoccurring) elements.

Example This pattern was applied in betsy for BPEL and BPMN expressiveness tests as the test suite is based on the workflow control-flow patterns [257, 286] which are created from analyzing multiple WfMSs. In BenchFlow, a large corpus of processes is used to construct workloads for performance tests [234, 237].

Pattern Stub Extension (P3)

Problem Create tests correctly (C2)

Solution Use a process stub (i.e., a minimal process which is extended for all tests) so that the extension solely contains the feature under test. The stub itself provides extension points, where the feature under test can be put. The rest is 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.

Example This pattern was applied in betsy for both, BPEL and BPMN.

Relations If the stub is fully functional, it can act as an Aptitude Test (P8).

Pattern Mutated Existing Test (P4)

Problem Create tests correctly (C2)

Solution Instead of starting from scratch, use correct tests and modify them by introducing a mutation [98]. This is especially useful for creating tests for faulty conditions: An existing test is mutated by injecting a single isolated fault, to see if a feature works correctly even in the face of errors [99].

Example The pattern was applied in betsy for creating erroneous processes that have to rejected upon deployment for both BPEL and BPMN. Also, it was applied to create a test suite for determining robustness [98].

Relations Similar to Stub Extension (P3) as the process model of another test is the basis for a new process test.

Pattern Open Sourcing (P5)

Problem Create tests correctly (C2), validate the procedure (C3), and validate the results (C8)

Solution Open source tests, procedure, and results, and put it under the scrutiny of the public. Public availability can help to find errors the original authors did not find. Also, this can help to build a community for the benchmark.

Example Both betsy and BenchFlow are open source. In the case of betsy, this has lead to contributions by experts and also engine vendors.

Relations May result in Expert Review (P6).

Pattern Expert Review (P6)

Problem Create tests correctly (C2), validate the procedure (C3), and validate the results (C8)

Solution Ask experts to review the benchmark artifacts. Experts can be domain experts, engine developers, or benchmark engineers.

Example For betsy, the maintainers of Apache ODE and bpel-g helped to improve the test cases through their feedback, looking at the results and checking why the behavior of their engine was different from what they expected.

Pattern Automatic Static Analysis (P7)

Problem Create tests correctly (C2), validate the procedure (C3), and validate the results (C8)

Solution Create static analysis checks which detect mistakes automatically. As most process languages are XML-based, an XML well-formedness check as well as schema validation with the XSD of the process language is straightforward. If possible, apply additional static analysis based on process language rules and best practices.

Example The pattern was applied in betsy by checking the correctness of processes regarding naming conventions, XML well-formedness, XSD validity regarding the process language schema, and even more sophisticated static analysis rules with BPELlint [101] and its equivalent for BPMN called BPMNspector [86].

8.4.2. Benchmarking Procedure Pattern

To *validate the procedure (C3)*, one can apply *Open Sourcing (P5)*, *Expert Review (P6)*, and *Aptitude Test (P8)*. *Reinstallation (P9)*, *Virtual Machines (P10)*, and *Containers (P11)* can be applied to *guarantee test isolation (C4)*. To *observe the process (C5)*, *Message Evaluation (P12)*, *Partner-based Message Evaluation (P13)*, *Execution Trace Evaluation (P14)*, *Engine API Evaluation (P15)*, *Concurrency Detection (P16)*, and *Detailed Logs (P17)* are applicable.

Pattern Aptitude Test (P8)

Problem Validate the procedure (C3) and determine the aptitude (C6)
Solution Define an aptitude test as a minimal requirement for participation in the benchmark. An engine must pass this test. The test should check the minimal amount of features required.
Example In betsy, there are two aptitude tests, one for BPEL named *Sequence*, containing a receive-assign-reply triplet (see Message Evaluation (P12)), and one for BPMN, named *SequenceFlow*, containing a start and end event, with corresponding script tasks to allow observing the events (see Execution Trace Evaluation (P14)), connected through sequence flows.
Relations Can be used as a stub for Stub Extension (P3).

Pattern Reinstallation (P9)

Problem Guarantee test isolation (C4)
Solution Install and start the engine anew for each test case, providing a fresh engine instance. Although a reinstallation can be time-consuming, it ensures that one test case cannot interfere with another one.
Example In betsy, this is the default mode.
Relations Virtual Machines (P10) and Containers (P11) are alternatives.

Pattern Virtual Machines (P10)

Problem Guarantee test isolation (C4)
Solution Create a Virtual Machine (VM) with a snapshot of a running engine upfront. This may require some time and effort once per engine. With, however, a snapshot in place, each test can be executed in isolation. The snapshot can easily be restored before each test and be discarded afterward, resulting in test isolation with a low temporal overhead. However, for VMs, there is typically a substantial RAM and HDD overhead.
Example Since 2014, betsy also supports this pattern [100].
Relations Reinstallation (P9) and Containers (P11) are alternatives.

Pattern Containers (P11)

Problem Guarantee test isolation (C4)

Solution Create an image with the engine already installed and configured. Create a new container for each test and discard the container afterward, effectively ensuring test isolation. This is similar to Virtual Machines (P10), but with considerably less overhead. At this point in time, however, support for RAM snapshots of containers is not existing, but HDD snapshots are available.

Example This pattern is used in both, betsy [84] and BenchFlow [62].

Relations Reinstallation (P9) and Virtual Machines (P10) are alternatives.

Pattern Message Evaluation (P12)

Problem Observe the process (C5)

Solution Send messages to the process and compare responses with an expected response. Use small interfaces with only few methods to keep different message types and possibilities low.

Example Betsy communicates with BPEL instances only through four different SOAP messages and observes the behavior by checking the responses. Under the hood, betsy uses soapUI.

Relations Partner-based Message Evaluation (P13) builds upon this pattern. Execution Trace Evaluation (P14) and Engine API Evaluation (P15) are alternatives.

Pattern Partner-based Message Evaluation (P13)

Problem Observe the process (C5)

Solution The process under test sends messages to an external service which the benchmarking system controls. The calling pattern of the service can be checked and compared to the expected interaction.

Example In betsy, this pattern is used to mock any partner service a BPEL process is required to communicate with. Moreover, concurrency detection was implemented with a mocked partner service as well.

Relations This pattern is an extension of Message Evaluation (P12). Alternatives are Execution Trace Evaluation (P14) and Engine API Evaluation (P15). Concurrency Detection (P16) can be implemented using this pattern.

Pattern Execution Trace Evaluation (P14)

Problem Observe the process (C5)

Solution The process writes log traces to the disk. The benchmarking framework then reads the log traces and compares them with expected ones. Use a small set of different standardized log traces. One can even inspect Detailed Logs (P17) and convert log statements to log traces.

Example This pattern is used in process mining, but also in betsy for observing the behavior of BPMN processes, as Message Evaluation (P12) does not work because of the lacking support for sending and receiving messages. In script tasks, log traces are written to a log. 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 process did exit correctly.

Relations Concurrency Detection (P16), Message Evaluation (P12), Partner-based Message Evaluation (P13), Engine API Evaluation (P15), and Detailed Logs (P17)

Pattern Engine API Evaluation (P15)

Problem Observe the process (C5)
Solution Use the API provided by the engine to query the deployment state of the process model and the current state as well as the history of specific process instances. As this is engine-dependent, it profits from applying Engine Layer Abstraction (P18) as well.
Example In both betsy and BenchFlow, the BPMN engines are queried about their deployment and final states. In BenchFlow, the correctness of the process instances are solely determined via the API of the engine.
Relations Alternative to Partner-based Message Evaluation (P13), Message Evaluation (P12), and Execution Trace Evaluation (P14), but works fine together with Engine Layer Abstraction (P18).

Pattern Concurrency Detection (P16)

Problem Observe the process (C5)
Solution Identify the parallel branches in the process under test. Upon entering and exiting each branch, a timestamp has to be stored alongside a branch ID at runtime. If the enter-exit pairs of parallel branches overlap, concurrency (being either real concurrency or at least nondeterministic interleaving) has been detected. The concurrency traces can either be tracked through an an external service or a log.
Example Betsy applies this pattern relying on Partner-based Message Evaluation (P13) for BPEL and Execution Trace Evaluation (P14) with a separate concurrency detection log for BPMN.
Relations Can be used either with Partner-based Message Evaluation (P13) or Execution Trace Evaluation (P14).

Pattern Detailed Logs (P17)

Problem Observe the process (C5)
Solution Configure the engine to use verbose logging. Otherwise, it might not be possible to observe everything that is important regarding the state of a process.
Example In betsy, detailed logs are enabled for several engines by replacing the log configuration file with a more verbose one.
Relations Execution Trace Evaluation (P14) & Concurrency Detection (P16).

8.4.3. Engine Pattern

To *determine the aptitude (C6)*, one can apply the *Aptitude Test (P8)*. *Engine Layer Abstraction (P18)*, *Failable Timed Action (P19)*, *Timeout Calibration (P20)*, and *Detailed Logs (P17)* can be used to *interact with the engines (C7)*.

Pattern Engine Layer Abstraction (P18)

Problem Interact with the engines (C7)

Solution Create 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, installation, deployment, starting, and other engine-specific assertions such as how to behave after an abortion of a process.

Example The Uniform BPEL Management Layer (UBML) [97] has been extracted from betsy. UBML is an engine-independent layer to (un)install, start, and stop the engine as well as to deploy processes and collect logs.

Relations Can rely upon the three different pattern candidates Failable Timed Action (P19), Timeout Calibration (P20), & Detailed Logs (P17). Eases Reinstallation (P9).

Pattern Failable Timed Action (P19)

Problem Interact with the engines (C7)

Solution The test system executes a specified action. Then it waits for a specific period during which success and failure conditions are checked every X milliseconds. The action fails if time is exceeded or if failure condition is met. It succeeds if success condition is met within the specific period.

Example As most engines do not support a synchronous API, betsy needs to rely on Failable Timed Action (P19). The act of deploying a process often involves copying the artifact to a specific location on the File System, after which the engine deploys it automatically, and then evaluating success through log inspection (see Detailed Logs (P17)).

Relations May require Detailed Logs (P17), should be used with Timeout Calibration (P20).

Pattern Timeout Calibration (P20)

Problem Interact with the engines (C7) and validate the results (C8)

Solution Before an actual machine is used for benchmarking, calibrate the timeouts that are required in the tests itself, in a Failable Timed Action (P19), or for Reinstallation (P9). The calibration of timeouts in the tests is necessary, in case a test produces nondeterministic results depending on different timeout settings.

Example Betsy implements a mechanism using the Aptitude Test (P8) to calibrate typical timeout values with a security range.

Relations Ensures that the timeouts in Reinstallation (P9) and Failable Timed Action (P19) are suitable.

8.4.4. Results Pattern

To *validate the results (C8)*, one can apply *Open Sourcing (P5)*, *Expert Review (P6)*, *Aptitude Test (P8)*, *Timeout Calibration (P20)*, and *At Least One Success (P21)*.

Pattern At Least One Success (P21)

Problem Validate the results (C8) and create tests correctly (C2)

Solution Compare the benchmarking results of one test between engines. If the test does not succeed on at least one engine, it is necessary to investigate the test itself, since the test or the testing procedure might be broken.

Example In betsy, this pattern was applied multiple times to detect issues in the tests, the procedure and the interaction with engines.

8.5. Discussion

The discussion about the pattern candidates is subdivided into three parts. First, the relationship between the pattern candidates and the challenges is put under scrutiny, answering the question which pattern candidate is a solution to a single challenge and which pattern candidate can be used as a solution to multiple challenges. The more (less) challenges a pattern candidate can be applied to, the more general (specific) it is. Second, the relationship between the pattern candidates themselves is evaluated, answering the question which pattern candidates are the basic building blocks of the pattern candidates language, and which are composite pattern candidates. The more a pattern candidate is used or required by other patterns, the more important it is for this pattern candidate catalog. Third and last, the different challenges and their groups are put under scrutiny regarding how well each pattern candidate fits into these groups or whether the pattern candidates are suitable for the whole domain, and not for a subdomain only.

General vs. Specific

There are four independent graphs in Figure 8.2. Three of them contain only a single challenge with multiple pattern candidates, whereas the other one contains multiple challenges with multiple pattern candidates. The more challenges a pattern candidate is the solution to, the darker the gray filling. Three pattern candidates, namely, Automatic Static Analysis (P7), Expert Review (P6), and Open Sourcing (P5), are solutions to the same three challenges, namely, create tests correctly (C2), validate the procedure (C3), and validate the results (C8). This makes sense as the three challenges are about validation or ensuring that no validation is required, which is independent of the target of the validation. In this pattern candidates catalog, these three pattern candidates are the most generic ones and are not that related to the domain of process engine benchmarking. Nevertheless, they have been kept in the pattern candidates catalog as they were paramount to this work in practice. Furthermore, there are another three pattern candidates, namely, At Least One Success (P21), Aptitude Test (P8), and Timeout Calibration (P20) which are solutions to two challenges each. Each of them has a primary challenge it relates to, and a secondary one where it can help. For instance, the Aptitude Test (P8) primarily helps to determine the aptitude (C6), but it can be used to validate the procedure (C3) as well. The procedure is valid to some degree if the

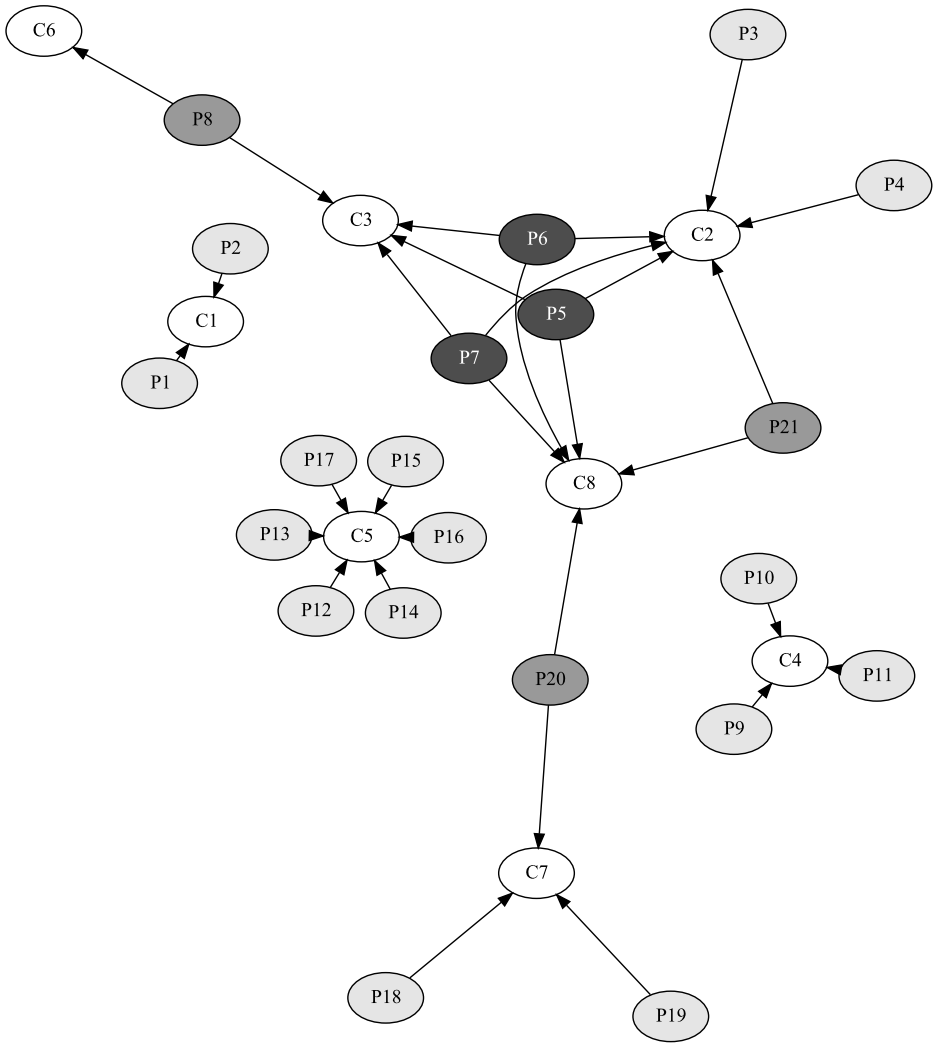


Figure 8.2.: Relationship Between Pattern Candidate and Challenge

Aptitude Test (P8) can be conducted via the full test procedure successfully. All remaining pattern candidates are solutions to a single challenge and considered to be specific to the process engine benchmarking domain. To sum up, from the 21 pattern candidates, 18 can be considered specific to the process engine benchmarking domain, although three of them are more generic but applicable nevertheless.

Basic vs. Composite

The 21 pattern candidates are related. Their relationships can be classified into the following four groups: *alternatives*, *requirements*, *facilitators*, and *implementations*[106]. In Figure 8.3, these relationships are depicted. Each pattern candidate 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 candidate is an alternative for each other.

[106] An *implements with* relationship is a sign of having different levels of abstraction. Only Concurrency Detection (P16) is a candidate for a higher level pattern. Hence, the different layers of abstraction is neglected in the discussion, and the focus is solely put on the composition characteristics.

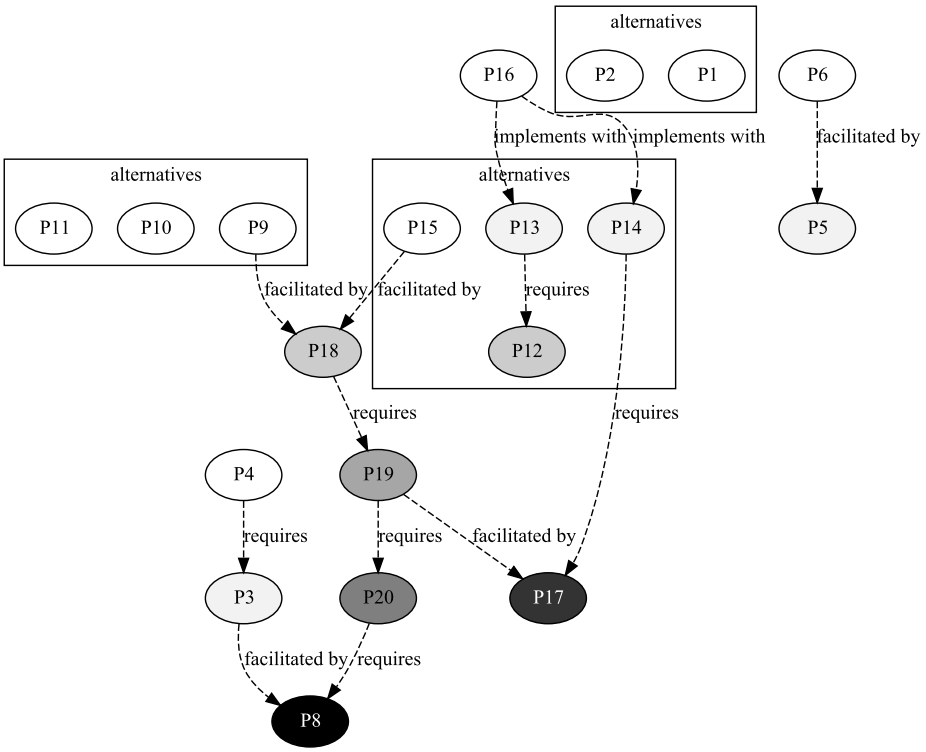


Figure 8.3.: Relationships Between Pattern Candidates

This has been done to reduce the amount of overlapping edges, and make the representation of the relationships between the pattern candidates easier to comprehend. Also, the more pattern candidates one pattern candidate is relied upon, the darker is the gray the node is filled with. Two pattern candidates, namely, Automatic Static Analysis (P7) and At Least One Success (P21), are singled out as they do not have a relationship with any other pattern candidates. The other 19 pattern candidates, however, form a language of pattern candidates. Three groups of pattern candidates represent one particular problem for which these pattern candidates are alternative solutions to. Normally, there is only a single pattern candidate for a specific problem. The most important pattern candidates are Aptitude Test (P8), Detailed Logs (P17), Timeout Calibration (P20), Failable Timed Action (P19), and Engine Layer Abstraction (P18), as they are directly and transitively related to seven, six, four, three, and two other pattern candidates, respectively, not taking into account alternative relationships. Together, they enable six other pattern candidates, summing up to eleven pattern candidates which are more than the half of the pattern candidate catalog. Taking the alternative relationships into account, the largest connected pattern candidates graph comprises 15 pattern candidates.

Challenges and Subdomains

The challenges are structured into subdomains of the process engine benchmarking domain. When looking at both, Figure 8.2 and Figure 8.3, it becomes clear that only a few pattern candidates can be attached to a single subdomain. The three challenges identify the tests (C1), guarantee test isolation (C4), and observe the process (C5) first seem to group disjunct patterns, but when tak-

ing the relationships between the patterns into account, they are becoming related as well. Hence, the pattern candidates cannot be split into separate sets dependent on the challenge subdomains.

8.6. Summary

This chapter reiterated the four process benchmarking aspects (i.e., tests, benchmarking procedure, process engines, and benchmarking results) with their seven major challenges. The typical solutions used throughout this work have been condensed into 21 Process Engine Benchmarking Pattern Candidates. They cover the kernel of this work in an easy to reuse and shareable format. Together, they form a pattern language. Hence, this supports hypothesis H4.6 *("Patterns are a suitable form to describe the central elements of process engine benchmarking.")*.

Future Work

In the future, it is planned to refine, revise and extend these 21 pattern candidates with domain experts. The goal is a pattern language which establishes itself as a common vocabulary in the domain of process engine benchmarking. Furthermore, it is planned to evaluate whether these pattern candidates of the process engine benchmarking domain can be generalized to the whole benchmarking domain, and therefore increasing their impact.

Part III.

Related Work and Conclusion

> The best way to predict the future is to create it.
>
> *Peter Drucker*

9. Conclusion and Outlook

This work targets the issue of providing information about the quality of process engines so that those can be compared within selection decisions because the available information is not sufficient for such tasks. To solve this, an automated benchmarking framework is proposed that can reveal such objective and ascertained information. In this third and last part, a summary of the contributions is presented first in Section 9.1, followed by a reiteration of the competing approaches to position the contributions among its competitors in Section 9.2. Last, the most critical limitations are stated along with open problems in this field of research in Section 9.3. Structure

9.1. Summary of Contributions

Ten IT artifacts have been designed and evaluated using the design science methodology by Hevner et al. [108] in the effort to support hypothesis H4 *("With automated benchmarking, it is possible to retrieve objective and comparable information on the quality characteristics of widely different process engines reproducibly and efficiently.")*. Those artifacts and the way they have been evaluated are summarized in the following. Ten IT Artifacts

Process Engine Abstraction Layer (PEAL) and PEAL Prototype: PEAL is a uniform API to interact with varying process engines consistently. The API has been evaluated by walking through the scenario of enabling and disabling the execution of a process model which is required for process engine benchmarking. The results showed that the methods are sufficient to cover those scenarios. Also, PEAL has been evaluated through a prototype that implements the mappings to three BPMN and seven BPEL process engines in various versions and configurations. All of those process engines fulfilled the previously mentioned scenario during testing. PEAL

Process Engine Benchmark Language (PEBL) PEBL is a DSL for expressing benchmarks and their results. It includes a representation of those benchmarks in XML and JSON. The language has been evaluated by checking how well eight benchmarks for two different process languages (i.e., BPEL and BPMN) and four different quality characteristics (i.e., functional suitability, usability, performance efficiency, and resilience) can be expressed. Results showed that the expressiveness fits and the generic script-based extension points are seldom required. PEBL

PEBWORK **Process Engine Benchmark Framework (PEBWORK) and the corresponding prototype BPEL/BPMN Engine Test System (betsy)** PEBWORK is a framework for conducting benchmarks and producing high-quality results. It includes a benchmark procedure comprising the required activities for a benchmark execution. PEBWORK builds upon PEAL and PEBL. It is evaluated by checking how the framework with its benchmark DSL and uniform engine API helps for designing and executing good benchmarks. Results show that multiple of the criteria for good benchmarks are automatically guaranteed upon using that framework for benchmarking. On top, the framework is implemented in betsy, which in turn is evaluated by conducting seven different kinds of benchmarks. It has been shown that it is feasible to automate PEBWORK and that the results are of the expected quality.

PEBDASH **Process Engine Benchmarking Interactive Dashboard (PEBDASH)** The PEBDASH is an interactive dashboard that presents the benchmark results so that the user can find the relevant information quickly. It is driven by eleven requirements in the form of user stories that are collected iteratively by domain experts in a small study. In the evaluation, it is shown that those requirements are met in the implementation of the dashboard.

PEBWORK **Efficient Process Engine Benchmark Framework (ePEBWORK) and the prototype virtualization-enabled betsy (vbetsy)** ePEBWORK is a more efficient version of PEBWORK. It builds upon the concept of restoring snapshots of VMs to provide fresh instances of process engines in constant time independent on the normal installation and startup durations. The evaluation is performed by comparing the time to result of betsy with that of vbetsy. Results showed that vbetsy is up to 94% faster than betsy for a single test execution.

PEBPATT **Process Engine Benchmarking Pattern Candidates (PEBPATT)** PEBPATT comprise 21 pattern candidates that are grouped according to four challenges in the domain of process engine benchmarking. The pattern candidates have been evaluated by peer-review of experts from three different universities and by analyzing their relationship with each other and with their challenges. The analysis showed that the pattern candidates form a consistent preliminary pattern language.

Benchmarks **Benchmarks** Out of the eight benchmarks with which PEBL has been evaluated, seven are performed with betsy, and six are contributions of their own as part of this work. Those six are benchmarks for evaluating the functional suitability, resilience, and usability of BPEL engines and the functional suitability of BPMN engines. They are evaluated by peer-review, execution, and result analysis. It has been shown that the benchmarks provide useful insights into the quality characteristics of the evaluated process engines. A summary of their results is found in Section 5.5.2.

Summary In summary, it has been shown that it is possible to retrieve and present information about the quality characteristics of widely different process engines reproducibly through PEAL, PEBL, PEBWORK, and PEBDASH *effectively*.

Table 9.1.: Summary of Theoretical Evaluation according to the Good Benchmark Criteria

Criteria	PEAL	PEBL	PEBWORK	ePEBWORK	PEBDASH	Σ
Affordable	+	+	+	+		+
Relevant						
Portable	+	+	+			+
Accessible		+	+		+	+
Clear		+				+
Solvable						
Scalable	+			+		+
Repeatable	+	+	+	+		+
Verifiable	+	+	+		+	+

Through ePEBWORK, that can even be done *efficiently*. As shown in Table 9.1, those contributions enable the better fulfillment of seven out of the nine criteria for a good benchmark. That knowledge about process engine benchmarking is captured for the process engine benchmarking community in the form of PEBPATT. Hence, hypothesis H4 is supported, which concludes this work.

9.2. Competing Approaches

In the previous chapters, related and competing work corresponding to the concepts within those chapters have already been discussed. In this section, the comprehensive and holistic approaches that compete with this work are described. The three most relevant holistic competing approaches to this work are, in decreasing order of competitiveness, BenchFlow [61, 62, 195, 232, 234, 236, 237], the approach by Delgado et al. [45], and BPELUnit [161, 168] as the representative of the set of highly similar WS testing approaches. In the following, these three competing approaches are detailed followed by a summary of related approaches which have already been detailed in the previous chapters.

Competition

The aim of the project *BenchFlow*[107] is similar to this work as BenchFlow aims to benchmark the performance of process engines implementing BPMN [61, 195, 232, 234–236]. Its primary goal is to reveal performance bottlenecks. Hence, in the project BenchFlow *only* the performance of process engines supporting BPMN are evaluated using a benchmark method [62], a standardized workload [233, 235] that represents typical real world scenarios, and a benchmarking framework [61, 62]. Its benchmark method describes the interaction with vendors about publishing the results, which is not covered by this work. Nevertheless, this work could benefit from such a benchmark method when

BenchFlow

[107] See `http://www.iaas.uni-stuttgart.de/forschung/projects/benchflow.php` and `http://design.inf.usi.ch/research/projects/benchflow`, visited 2017-3-31

aiming to extend PEBDASH with quality characteristics of proprietary process engines as well, as it may help to get vendor approval more easily. Skouradaki et al. [235] propose to design workloads by detecting reoccurring structures within real-life BPMN process models. Designing a workload is specific to performance benchmarking and not necessary for the other quality characteristics, and therefore this work does not cover that except that workloads can be specified using extension attributes in PEBL. As part of the framework evaluation [237], workloads are also designed using the workflow patterns [257]. The focus, however, has been, of course, on how well the engines execute those workflow patterns and not on how many of those patterns are supported. But to design such a workload, the results of this work would help. For instance, knowing which language constructs are supported by all BPMN engines determines the set of language constructs with which a portable workload can be constructed. The aspect that shares the most with this work is the BenchFlow benchmarking framework. It also uses abstraction layers to communicate uniformly with the engines, serializes the benchmarks in its own DSL from which the actual test bed is generated (i.e., a model-driven approach), and is able to calculate Key Performance Indicators (KPIs) (i.e., aggregated metrics) from the gathered test results with their atomic measurements. The abstraction layer is categorized into core APIs that correspond to the process model and process instance services of PEAL and non-core ones comprising tasks such as creating users, claiming tasks, or issuing events which are not available in PEAL. Moreover, they use a uniform log representation of the engine-specific logs to extract performance data, similar to the engine-specific log trace extractions in PEBWORK. As shown in Section 4.5.3.4, the benchmark DSL in this work can represent the benchmarks and results defined in the BenchFlow DSL. A BenchFlow-agnostic DSL [236] of performance benchmarking of BPMS in general is available, too, and can be seen as a superset of the BenchFlow DSL. The framework itself builds upon the distributed performance testing framework Faban and makes use of Docker to ensure a frozen infrastructure. Hence, it provides a setup that ensures reproducibility, and therefore enables performing good benchmarks [111, 229]. Instead of building upon containers, ePEBWORK builds upon VMs to quickly restore previously created snapshots for quick test bed creation. Although PEBL can express performance benchmarks, PEBWORK can only execute them in theory. To sump up, this work evaluates both BPEL and BPMN-based process engines by a variety of different quality characteristics in contrast to BenchFlow that focuses only on performance benchmarking of BPMN engines. Hence, while this work is more generic than BenchFlow, they complement each other. This work can build upon the performance benchmark execution prototype of BenchFlow as well as the workload creation methods.

Delgado et al.

Delgado et al. [45] propose a list of 94 relevant key characteristics (79 technical and 15 nontechnical extracted from a variety of reports and RfI) to evaluate process engines and a method to systematically evaluate BPMS

through test cases for their key characteristics. Their method and list of key characteristics is evaluated in a case study covering 19 open source and proprietary BPMS supporting BPMN, BPEL, or XPDL. This sounds similar to this approach at first, but looking more closely, there is a variety of huge differences which are detailed subsequently. First, Delgado et al. [45] focus on BPMS and take both inherent and assigned quality attributes (e.g., workflow pattern support but also price) into account. This is in contrast to this work which focuses solely on inherent quality attributes of the heart of BPMS, namely, the process engine. Moreover, in this work, also nonfunctional characteristics (e.g., performance and usability) are evaluated whereas Delgado et al. do not cover those. Second, in their approach, the key characteristics (i.e., evaluation criteria) have to be weighed with trivalent priorities (mandatory, medium priority, low priority) and the support for those characteristics is measured with a trivalent scale (total, partial, or no support) which can be further specified according to a level of compliance (native, particular, and integration) specifying any additional effort required to achieve the specified support. It is, however, unclear how these three scores influence the final score of each BPMS as no formula is given. In this work, the information relevant for such a decision is provided, and the decision maker can resort to any method he deems fit (e.g., AHP). In other words, Delgado et al. [45] focus on business effectiveness and this work focuses on technical effectiveness instead (see Section 1.3). Moreover, it is not proven that their method of guiding the selection decision is superior in comparison to other thoroughly evaluated MCDM methods such as AHP. Furthermore, the support is given in a trivalent fashion, losing a lot of detail for the sake of that abstraction. This work supports metrics with arbitrary result formats and aggregation hierarchies. Third, although they use theoretical and practical test cases, no representation format or automated execution (i.e., tool support or execution guidance) for the practical test case is provided. It is, therefore, assumed that these test cases are represented in an unstructured way and conducted manually with the possibility of human error. In addition to the absence of tool support for the practical test case execution, the whole method lacks tool support whereas this work is fully automated through multiple tools working hand in hand. This shows that Delgado et al. focus on the selection decision instead on the reproducibility of the test results. Fourth, their methodology is specified on a high level in an abstract way without any guidance how to actually evaluate the characteristics. It lacks a framework to ensure test automation, test isolation, and automated measures to detect flaws in the evaluation. In comparison, this work provides test case representation, arbitrary metrics, and automated benchmarking. In summary, both Delgado et al. [45] and this work have a different focus but would complement each other perfectly as they are highly related and interoperable.

Another competitive approach is BPELUnit [161, 168] which is a xUnit-like framework to automatically execute tests against BPEL processes. It comprises

BPELUnit

adapters that can handle automated deployment and undeployment or following protocols (e.g., through SOAP header fields), an XSD-based test model to describe the test cases, and a prototype that can automatically execute XML-based test cases. The deployment adapters are, functionality-wise, a small subset of PEAL because these adapters do not generate deployment descriptors or handle more than the actual deployment and undeployment steps. In contrast, PEBL does not provide protocol adapters, but fields within the SOAP header could nevertheless be set on the SOAP message level or through scripts. The test model has some overlap with PEBL: they represent test cases with their steps and assertions as well as test partners similarly. BPELUnit even provides support for asynchronous request/response pairs. But the test model of BPELUnit does not store engine-specific logs nor does it allow to specify metric aggregations, making clear that its focus is a single test for a process. Although PEBL offers the additional possibility to serialize itself in JSON as well, it lacks tool support for editing any serialization in contrast to BPELUnit with its Eclipse-based UI. In comparison, this work covers both BPEL and BPMN, the API for the engines is not only about deployment of process models but also about the lifecycle of the engine and the management of instances, and last, does not guarantee full test isolation in between the tests as no fresh engine is provided, hindering the support for good benchmarks [111]. This approach is representative of a variety of other testing approaches that build upon the WSDL standard interfaces, such as SOABench [17], GENESIS2 [123], TASSA [198], and soapUI. This work builds upon soapUI for performing the actual test step instead of reinventing the wheel for BPEL, although BPELUnit could have been chosen as well. For BPMN, there are no similar tools available. BPELUnit builds upon and is motivated by their stated BPEL testing architecture consisting of four layers that build upon another: test specification, organization, execution, and results layer. The differences and similarities can be seen according to these four layers, too. Both allow a data-centered (through predefined language constructs) and a logic-centered approach (through scripts) to design the test specification. And both fully automate the test execution, but while BPELUnit provides tool support for creating the tests, this work provides methods (C2FM and F2TM) for that task instead. Regarding the test organization, although both allow to structure the tests hierarchically, only BPELUnit allows to specify fixtures for a set of tests. Fixtures are stated implicitly in this work, ensuring test isolation for instance through providing fresh engine instances for each test. And last, both use real-life test execution.

Related

There are other related approaches which are similar in some way but reside in other domains. A small selection of those approaches is outlined in the following. In the domain of comparing process modeling tools, Geiger and Wirtz [79] as well as the BPMN Model Interchange Working Group (BPMN MIWG)[108]

[108] `https://github.com/bpmn-miwg`, visited 2017-3-31

have conducted studies. The BPMN MIWG even provides a dashboard[109] of their results and a white paper[110] on their approach. Instead of evaluating process engines, the process languages can be evaluated as well separately by Lu and Sadiq [158] and how well they fit for the varying purposes by Thöne et al. [251]. Instead of checking the conformance of a process engine to the standard, the conformance of the process model to some specification or with other process models is researched extensively [25, 76, 80, 258, 262].

9.3. Limitations and Open Problems

The limitations and open problems of this work can be subdivided into five different aspects. They start from the most straightforward open problems corresponding to technical effectiveness and execution efficiency over to limitations and open problems of sharing the produced knowledge and results to, finally, ideas how to use the presented approach in a wider scope such as microservices and the application of MCDM approaches for selecting BPMSs.

Technical Effectiveness

Although the process engine approach has been evaluated with three of the quality characteristics of the quality product model in the ISO/IEC standard 25010 [113], namely, functional suitability, usability, and resilience, there is still room for improvement. Only the DSL for expressing benchmarks has been evaluated for performance efficiency. In the future, the approach will be evaluated with more benchmarks that will cover the remaining quality characteristics. Work from Lenhard [148] will be taken into account for covering portability, and BenchFlow for covering performance efficiency fully. Furthermore, cloud-based process engines (e.g., Signavio Workflow[111]) will be incorporated in the process engine evaluation in the future as well. Because of the BPaaS model, the process engine evaluation approach needs to take the peculiarities of evaluating cloud services (e.g., Kolb and Wirtz [133], Schlauderer and Overhage [227]) additionally into account.

Execution Efficiency

In this work, the efficiency of the process engine evaluation has been improved by leveraging snapshots of VMs on a single machine sequentially. This opens up three possibilities for improvement that can be applied orthogonally. First, the overhead when working with VMs is not neglectable. An alternative is to use the less resource intensive container [22, 31, 60, 293]. Since the release of Docker Engine[112] 1.13 in 2017, it is possible to create RAM snapshots[113] for container instances leveraging the experimental feature which is known as “checkpoint and restore.” Second, the process engine evaluation could be

[109] http://bpmn-miwg.github.io/bpmn-miwg-tools/, visited 2017-3-31
[110] http://www.bptrends.com/bpt/wp-content/uploads/06-03-2014-ART-MakingBPM-a-TrueLinguaFranca-Zbigniew-Misiak-etal.pdf, visited 2017-3-31
[111] https://www.signavio.com/products/workflow/, visited 2017-3-31
[112] https://www.docker.com/products/docker-engine, visited 2017-3-31
[113] https://github.com/docker/docker/blob/master/CHANGELOG.md, visited 2017-3-31

migrated from a sequential to a parallel evaluation which is expected to provide additional performance improvements. By building upon containers, multiple containers could be running on the same machine as long as the resources suffice. Third, the test execution can be distributed onto a cluster of machines to reduce the time to results even further. This goes hand in hand with the support for containers as containers can be easily distributed across a cluster of machines, e.g., through Docker Swarm[114].

Sharing is Caring

The 21 Process Engine Benchmarking Pattern Candidates (PEBPATT) are a start towards capturing best practices and solutions to challenges within the domain of process engine benchmarking. In the future, those pattern candidates need to be converted to full patterns (i.e., add context and forces) within a larger (i.e., add more patterns) pattern language by following the "rule of three" [6] (i.e., check which patterns are used by BenchFlow or other process engine benchmarking approaches as well). An extended version of those patterns is currently under review at the EuroPLoP[115] conference [103]. Besides sharing the knowledge about the challenges and solutions in process engine benchmarking, the benchmarks and their results shall be shared as well. The platform that is based on the presented dashboard is currently limited to benchmarks and results produced as part of this work. In the future, this platform shall be the place to go looking for benchmark results about process engines and to put existing (e.g., by Delgado et al. [45], Bianculli et al. [16], and Wohed et al. [289]) and new benchmarks and their results. The next step would be to provide a benchmarking platform which will automatically conduct any benchmarks and publish the results in the dashboard for any integrated process engine online. A summary of the lessons learned for practitioners based on the experience on process engine benchmarking is currently under review [63, 151] as well.

Micro Engines

Centralized BPMSs are a bad fit for implementing distributed and modular architectures based on microservices [252]. The need for executing business processes, however, still perseveres and the solution to implement the business processes by hand through an object oriented programming language within the microservices is far from elegant. A solution can be to use stripped down process engines (i.e., the heart of BPMSs [278]) as shown by Nikol et al. [176] which has been supervised by the author of this dissertation. Such engines enable the execution of business processes within such microservices because they require fewer resources, start quicker, and are simpler to install. Therefore, the benefits of a distributed and modular IT architecture can be kept while executing processes natively. The artifacts of this work can be used to drive the development (e.g., through Continuous Integration) of such stripped-down process engines in the future by providing an infrastructure to ensure the quality of those products.

[114] `https://www.docker.com/products/docker-swarm`, visited 2017-3-31
[115] `http://europlop.net/`, visited 2017-3-31

Business Effectiveness

In this work, the focus has been on evaluating process engines, which are the heart of BPMSs. MCDM approaches such as AHP can be used for selecting process engines, but, typically, they are used for selecting BPMSs. In the future, this approach will be extended to cover BPMSs, including the modeling component as well by integrating with research by Geiger and Wirtz [79]. Moreover, the integration of the produced benchmark results with such MCDM approaches will be covered as well, providing an automated matching with the revealed quality attributes of the process engines and BPMSs to speed up the decision processes. This should let the decision makers focus on defining and prioritizing their requirements, leaving the other aspects such as matching and revealing the quality attributes to a machine instead.

Part IV.

Appendix

A. Engines under Test

In this work, both BPEL and BPMN process engines have been evaluated (i.e., put under test). An overview is given in the following for both BPEL and BPMN in Table A.1.

Table A.1.: Details of Benchmarked BPEL and BPMN Engines

Language	Name	Version	License	Developed in	Released at	Configuration
BPEL	Apache ODE	1.3.5	Apache-2.0	Java	2011-02-06	
BPEL	Apache ODE	1.3.5	Apache-2.0	Java	2011-02-06	in-memory
BPEL	Apache ODE	1.3.6	Apache-2.0	Java	2013-10-12	
BPEL	Apache ODE	1.3.6	Apache-2.0	Java	2013-10-12	in-memory
BPEL	OpenESB	2.2	CDDL-1.0	Java	2009-12-01	
BPEL	OpenESB	2.3	CDDL-1.0	Java	2011-02-01	
BPEL	OpenESB	2.3.1	CDDL-1.0	Java	2013-10-01	
BPEL	OpenESB	3.0.1	CDDL-1.0	Java	2015-02-13	
BPEL	OpenESB	3.0.5	CDDL-1.0	Java	2015-06-24	
BPEL	Orchestra	4.9	LGPL-2.1+	Java	2012-01-23	
BPEL	ActiveBPEL	5.0.2	GPL-2.0+	Java	2008-05-09	
BPEL	petalsesb	4.0	LGPL 2.1+	Java	2012-02-02	
BPEL	petalsesb	4.1	LGPL 2.1+	Java	2012-07-06	
BPEL	bpel-g	5.3	GPL-2.0+	Java	2012-04-27	
BPEL	bpel-g	5.3	GPL-2.0+	Java	2012-04-27	in-memory
BPEL	WSO2	2.1.2	Apache-2.0	Java	2011-10-30	
BPEL	WSO2	3.0.0	Apache-2.0	Java	2012-10-17	
BPEL	WSO2	3.1.0	Apache-2.0	Java	2013-12-06	
BPEL	WSO2	3.2.0	Apache-2.0	Java	2014-02-03	
BPEL	WSO2	3.5.1	Apache-2.0	Java	2016-02-29	
Language	**Name**	**Version**	**License**	**Developed in**	**Released at**	**Configuration**
BPMN	camunda BPM	7.0.0	Apache-2.0	Java	2013-08-31	
BPMN	camunda BPM	7.1.0	Apache-2.0	Java	2014-03-31	
BPMN	camunda BPM	7.2.0	Apache-2.0	Java	2014-11-28	
BPMN	camunda BPM	7.3.0	Apache-2.0	Java	2015-05-29	
BPMN	camunda BPM	7.4.0	Apache-2.0	Java	2015-11-30	
BPMN	camunda BPM	7.5.0	Apache-2.0	Java	2016-05-31	
BPMN	Activiti	5.15.1	Apache-2.0	Java	2014-04-01	
BPMN	Activiti	5.16.3	Apache-2.0	Java	2014-09-17	
BPMN	Activiti	5.17.0	Apache-2.0	Java	2014-12-18	
BPMN	Activiti	5.18.0	Apache-2.0	Java	2015-07-31	
BPMN	Activiti	5.19.0	Apache-2.0	Java	2015-11-05	
BPMN	Activiti	5.20.0	Apache-2.0	Java	2016-04-18	
BPMN	Activiti	5.21.0	Apache-2.0	Java	2016-06-13	
BPMN	Activiti	5.22.0	Apache-2.0	Java	2016-11-04	
BPMN	jBPM	6.0.1	Apache-2.0	Java	2014-05-14	
BPMN	jBPM	6.1.0	Apache-2.0	Java	2014-08-19	
BPMN	jBPM	6.2.0	Apache-2.0	Java	2015-03-09	
BPMN	jBPM	6.3.0	Apache-2.0	Java	2015-09-28	
BPMN	jBPM	6.4.0	Apache-2.0	Java	2016-04-19	
BPMN	jBPM	6.5.0	Apache-2.0	Java	2016-10-25	

B. Tags for the BPEL Static Analysis Rules

Table B.1.: Covered SA Rules Grouped by Tag, taken from [202, p. 5]

tag	rules	Σ
violation check		
node requirements	1, 3, 13, 15, 17, 24, 35, 36, 45, 47, 50, 53–54, 57, 62, 76, 78, 80, 91	19
choice	16, 17, 19, 20, 25, 32, 34, 47, 51, 52, 55, 59, 63, 80, 81, 83, 85, 90	18
uniqueness	2, 14, 18, 22, 23, 44, 64, 66–69, 76, 86, 92, 93	15
consistent redundancy	5, 11, 12, 34–37, 46, 48, 57, 58, 79, 86, 87	14
location	6–8, 61, 65, 70, 71, 79	8
definition resolution	10, 65, 86, 95	4
execution instructions	84, 88, 89, 95	4
control cycle detection	72, 82	2
target activities		
WSDL definitions	1, 2, 5, 10–14, 19, 20, 22, 45–48, 50, 53, 54, 58, 84, 87, 88	22
message activities	5, 10, 46–48, 50–55, 58, 59, 61, 63, 78, 84, 85, 87, 89, 90	21
process and scope	3, 18, 23, 44, 61, 78–80, 82, 83, 88, 91–93	14
message assignment activities	47, 48, 50–55, 58, 59, 63, 85, 87, 90	14
FCT handler activities	3, 6–8, 10, 70, 71, 78–81, 93	12
flow activities	64–72, 82	10
partner link activities	5, 10, 16–18, 35–37, 84	9
variable activities	10, 23–25, 34, 48, 58, 86, 90	9
XSD definitions	10,–14, 45	6
assignment activities	10, 32, 34–37	6
correlation activities	10, 44–46, 88	5
event handler activities	83, 86, 88, 89, 95	5
loop activities	62, 70, 76, 83	4
start activities	15, 57, 62	3

C. Artifacts

Following links last accessed 8th August 2017.

- Source code of PEAL prototype
  `https://github.com/uniba-dsg/betsy/tree/master/peal`
- Source code of both, betsy and vbetsy
  `https://github.com/uniba-dsg/betsy/tree/master/`
- Source code of PEBL prototype
  `https://github.com/uniba-dsg/betsy/tree/master/pebl`
- Source Code of the loader
  `https://github.com/uniba-dsg/betsy/tree/master/loader`
- PEBL Schemas
  `https://github.com/uniba-dsg/betsy/tree/master/pebl/src/main/resources/pebl`
- Docker image of betsy
  `https://hub.docker.com/r/simonharrer/betsy-docker/`
- VM images for vbetsy
  `https://lspi.wiai.uni-bamberg.de/svn/betsy/ova`
- Source code of the sprinkle-based provisioning scripts for VMs of vbetsy
  `https://github.com/uniba-dsg/betsy-engines`
- Process engines installer for betsy
  `https://lspi.wiai.uni-bamberg.de/svn/betsy/`
- Source code of dashboard
  `https://github.com/peace-project/dashboard`
- Public dashboard
  `https://peace-project.github.io/`
- Benchmarks and Results in public dashboard database
  `https://peace-project.github.io/data/pebl.json`

Bibliography

[1] S. Adam *et al.*, "BPM Suites," Fraunhofer IESE, Tech. Rep., 2013. (Cited on pages 8 and 9.)

[2] ——, "BPM Suites," Fraunhofer IESE, Tech. Rep., 2014. (Cited on pages 8 and 9.)

[3] K. Agarwal, B. Jain, and D. E. Porter, "Containing the Hype," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*. ACM, 2015, p. 8. (Cited on page 44.)

[4] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau *et al.*, *WS-BPEL extension for people (BPEL4People)*, 2007, v1. 0. (Cited on page 28.)

[5] C. Alexander, *A Pattern Language*. Oxford University Press, Aug. 1978. (Cited on pages 14, 29, and 168.)

[6] B. Appleton, "Patterns and Software: Essential Concepts and Terminology," 1998, accessed 30-January-2017. [Online]. Available: http://www.bradapp.com/docs/patterns-intro.html (Cited on page 190.)

[7] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software Performance Testing Based on Workload Characterization," in *Proceedings of the 3rd International Workshop on Software and Performance (WOSP)*. New York, NY, USA: ACM, 2002, pp. 17–24. (Cited on page 170.)

[8] D. Baker, D. Bridges, R. Hunter, G. Johnson, J. Krupa, J. Murphy, and K. Sorenson, "Guidebook to decision-making methods," Department of Energy, USA, Tech. Rep., Dec. 2001. (Cited on pages 6 and 23.)

[9] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede, "Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection," Queensland University of Technology, Faculty of IT, Australia, Tech. Rep., 2005. (Cited on page 30.)

[10] ——, "Service Interaction Patterns," in *3rd International Conference on Business Process Management*, Nancy, France, Sep. 2005, pp. 302–318. (Cited on page 30.)

[11] A. P. Barros, G. Decker, M. Dumas, and F. Weber, "Correlation Patterns in Service-Oriented Architectures," in *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Braga, Portugal, Mar. 2007, pp. 245–259. (Cited on page 30.)

[12] V. R. Basili, "The Experimental Paradigm in Software Engineering," in *Experimental Software Engineering Issues: Critical Assessment and Future*

Directions. Springer, 1993, pp. 1–12. (Cited on page 8.)

[13] V. Belton and T. Gear, "On a Short-coming of Saaty's Method of Analytic Hierarchies," *Omega*, vol. 11, no. 3, pp. 228–230, 1983. (Cited on pages 22 and 23.)

[14] T. Berners-Lee, "Long live the web," *Scientific American*, vol. 303, no. 6, pp. 80–85, 2010. (Cited on page 136.)

[15] R. S. Bhadoria, N. S. Chaudhari, and G. S. Tomar, "The Performance Metric for Enterprise Service Bus (ESB) in SOA System: theoretical underpinnings and empirical illustrations for information processing," *Information Systems*, Dec. 2016. (Cited on page 118.)

[16] D. Bianculli, W. Binder, and M. L. Drago, "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)*. New York, NY, USA: ACM, 2010, pp. 141–150. (Cited on pages 8, 9, 10, 67, 168, and 190.)

[17] ——, "SOABench: Performance Evaluation of Service-oriented Middleware Made Easy," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 301–302. (Cited on pages 67, 118, and 188.)

[18] D. Bimamisa, M. Müller, S. Harrer, and G. Wirtz, "Interactive Dashboard for Workflow Engine Benchmarks," in *PEAcE*, 2016. (Cited on pages 17, 18, and 134.)

[19] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, vol. 16, no. 03, pp. 80–85, May 2012. (Cited on page 50.)

[20] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," in *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC)*, ser. LNCS, vol. 8274. Springer Berlin Heidelberg, Dec. 2013, Demonstration, pp. 692–695. (Cited on page 154.)

[21] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, ser. Advanced Web Services. Springer, Jan. 2014, pp. 527–549. (Cited on pages 50 and 154.)

[22] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015. (Cited on pages 43, 155, and 189.)

[23] K. Böhmer and S. Rinderle-Ma, "A systematic literature review on process model testing: Approaches, challenges, and research directions," University of Vienna, Tech. Rep., Sep. 2015. (Cited on page 118.)

[24] E. Börger, "Approaches to Modeling Business Processes. A Critical Analysis of BPMN, Workflow Patterns and YAWL," *Software & Systems Modeling*, vol. 11, no. 3, pp. 305–318, 2012. (Cited on page 31.)

[25] A. Both and W. Zimmermann, "Automatic Protocol Conformance Checking of Recursive and Parallel BPEL Systems," in *European Conference on Web Services (ECOWS)*. Dublin, Ireland: IEEE, Nov. 2008, pp. 81–91. (Cited on page 189.)

[26] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and Verification in Service-Oriented Architecture: A Survey," *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261–313, Jun. 2012. (Cited on page 67.)

[27] P. W. Bridgman, *Dimensional analysis*. Yale University Press, 1922. (Cited on page 22.)

[28] D. M. Buede and D. T. Maxwell, "Rank Disagreement: A Comparison of Multi-criteria Methodologies," *Journal of Multi-Criteria Decision Analysis*, vol. 4, no. 1, pp. 1–21, 1995. (Cited on page 22.)

[29] N. B. Bui, L. Zhu, I. Gorton, and Y. Liu, "Benchmark Generation Using Domain Specific Modeling," in *Australian Software Engineering Conference (ASWEC)*. Institute of Electrical and Electronics Engineers (IEEE), Apr. 2007. (Cited on pages 67 and 68.)

[30] B. Bukovics, *Pro WF: Windows Workflow in .NET 4*. Apress, Jun. 2010. (Cited on page 5.)

[31] R. Chamberlain and J. Schommer, "Using Docker to Support Reproducible Research," Technical report, Invenshure, LLC. . figshare. 1101910, Tech. Rep., 2014. (Cited on pages 43, 155, and 189.)

[32] P. M. Chen and B. D. Noble, "When virtual is better than real [operating system relocation to virtual machines]," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 133–138. (Cited on page 41.)

[33] S.-J. Chen and C.-L. Hwang, *Fuzzy Multiple Attribute Decision Making Methods*. Springer, 1992. (Cited on page 22.)

[34] R. Chinnici, M. Hadley, and R. Mordani, *JSR 224: JavaTM API for XML-Based Web Services (JAX-WS) 2.0*, Apr. 2006. (Cited on pages 55 and 76.)

[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286. (Cited on page 41.)

[36] A. Colyer, G. Blair, and A. Rashid, "Managing complexity in middleware," in *Proceedings of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Boston, 2003, pp. 21–26. (Cited on page 153.)

[37] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*. Association for Computing Machinery (ACM), 2010. (Cited on page 155.)

[38] G. Copeland and D. Maier, "Making smalltalk a database system," *ACM SIGMOD Record*, vol. 14, no. 2, p. 316, Jun. 1984. (Cited on page 3.)

[39] I. D. Craig, *Virtual Machines*. Springer, 2006. (Cited on page 41.)

[40] S. Dashevskyi, D. R. Dos Santos, F. Massacci, and A. Sabetta, "TESTREX: a testbed for repeatable exploits," in *Proceedings of the 7th USENIX conference on Cyber Security Experimentation and Test*. USENIX Association, 2014, pp. 1–1. (Cited on pages 43 and 155.)

[41] I. Davies and M. Reeves, "BPM Tool Selection: The Case of the Queensland Court of Justice," *Handbook on Business Process Management*, vol. 1, pp. 371–392, 2014. (Cited on pages 8, 9, and 10.)

[42] G. Decker and J. Mendling, "Process Instantiation," *Data and Knowledge Engineering*, vol. 68, pp. 777–792, 2009. (Cited on page 54.)

[43] G. Decker, O. Kopp, F. Leymann, and M. Weske, "BPEL4Chor: Extending BPEL for Modeling Choreographies," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*, Salt Lake City, Utah, USA, Jul. 2007, pp. 296–303. (Cited on page 28.)

[44] G. Decker, O. Kopp, and A. P. Barros, "An Introduction to Service Choreographies," *it – Information Technology, Oldenbourg Wissenschaftsverlag*, vol. 50, no. 2, pp. 122–127, 2008. (Cited on page 2.)

[45] A. Delgado, D. Calegari, P. Milanese, R. Falcon, and E. García, "A Systematic Approach for Evaluating BPM Systems: Case Studies on Open Source and Proprietary Tools," in *Open Source Systems: Adoption and Impact*, ser. IFIP Advances in Information and Communication Technology. Springer International Publishing, 2015, vol. 451, pp. 81–90. (Cited on pages 6, 7, 8, 9, 11, 118, 151, 185, 186, 187, and 190.)

[46] A. Delgado, D. Calegari, and A. Arrigoni, "Towards a Generic BPMS User Portal Definition for the Execution of Business Processes," *Electronic Notes in Theoretical Computer Science*, vol. 329, pp. 39–59, Dec. 2016. (Cited on page 50.)

[47] G. Din, K.-P. Eckert, and I. Schieferdecker, "A Workload Model for Benchmarking BPEL Engines," in *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW)*, Apr. 2008, pp. 356–360. (Cited on page 170.)

[48] DMTF, *Open Virtualization Format Specification*, Aug. 2015, v2.1.1. (Cited on pages 157 and 159.)

[49] B. F. V. Dongen, "A Meta Model for Process Mining Data," in *In Proceedings of the CAiSE WORKSHOPS*, Porto, Portugal, 2005, pp. 309–320. (Cited on page 65.)

[50] T. Dornemann, E. Juhnke, and B. Freisleben, "On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud," in *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2009, pp. 140–147. (Cited on page 51.)

[51] S. A. W. Drew, "From knowledge to action: the impact of benchmarking on organizational performance," *Long Range Planning*, vol. 30, no. 3, pp. 427–441, Jun. 1997. (Cited on page 37.)

[52] M. J. Duftler, N. K. Mukhi, A. Slominski, and S. Weerawarana, "Web Services Invocation Framework (WSIF)," in *OOPSLA Workshop on Object Oriented Web Services*, vol. 194, 2001. (Cited on page 49.)

[53] M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede, *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley, 2005. (Cited on pages 2 and 3.)

[54] Dunie *et al.*, *Magic Quadrant for Intelligent Business Process Management Suites*, Gartner, Aug. 2016. (Cited on pages 5, 8, and 9.)

[55] D. Eastlake, 3rd and P. Jones, *US Secure Hash Algorithm 1 (SHA1), RFC3174*, United States, 2001. (Cited on page 143.)

[56] T. M. Egyedi, "Standard-compliant, but incompatible?! ," *Computer Standards & Interfaces*, vol. 29, no. 6, pp. 605–613, 2007. (Cited on pages 5 and 10.)

[57] M. Elias and A. Bezerianos, "Exploration views: Understanding Dashboard Creation and Customization for Visualization Novices," in *Proceedings of the 13th International Conference on Human-Computer Interaction (INTERACT)*. Lisabon, Portugal: Springer, Sep. 2011, pp. 274–291. (Cited on page 136.)

[58] M. S. Feldman and J. G. March, "Information in Organizations as Signal and Symbol," *Administrative Science Quarterly*, vol. 26, no. 2, p. 171, Jun. 1981. (Cited on page 10.)

[59] M. Felleisen, "On the expressive power of programming languages," *Science of Computer Programming*, vol. 17, no. 1-3, pp. 35–75, Dec. 1991. (Cited on page 30.)

[60] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 28. Institute of Electrical and Electronics Engineers (IEEE), Mar. 2014, p. 32. (Cited on pages 42, 155, and 189.)

[61] V. Ferme, A. Ivanchikj, and C. Pautasso, "A Framework for Benchmarking BPMN 2.0 Workflow Management Systems," in *13th International Conference on Business Process Management (BPM)*. Innsbruck, Austria: Springer, Aug. 2015. (Cited on pages 51, 67, 68, 110, 111, 115, 118, 134, 142, and 185.)

[62] V. Ferme, A. Ivanchikj, C. Pautasso, M. Skouradaki, and F. Leymann, "A Container-centric Methodology for Benchmarking Workflow Management Systems," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. Scitepress, 2016. (Cited on pages 51, 68, 118, 168, 175, and 185.)

[63] V. Ferme, J. Lenhard, S. Harrer, M. Geiger, and C. Pautasso, "Lessons Learned from Evaluating Workflow Management Systems," in *ICSE*, 2017, Poster Track. (Cited on pages 17 and 190.)

[64] S. Few, "Dashboard Confusion," *Intelligent Enterprise*, Mar. 2004. (Cited on pages 135 and 136.)

[65] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," The Internet Society, RfC, 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt (Cited on page 99.)

[66] J. Figueira, S. Greco, and M. Ehrgott, *Multiple Criteria Decision Analysis: State of the Art Surveys*. Springer Science & Business Media, 2005, vol. 78. (Cited on page 22.)

[67] J. Figueira, V. Mousseau, and B. Roy, "ELECTRE methods," in *Multiple criteria decision analysis: State of the art surveys*. Springer, 2005, pp. 133–153. (Cited on page 22.)

[68] P. C. Fishburn, "Additive Utilities with Incomplete Product Set: Applications to Priorities and sharings," *Operations Research Society of America*, 1967. (Cited on page 22.)

[69] M. Fowler, "DomainSpecificLanguage," 2008, accessed 30-January-2017. [Online]. Available: https://martinfowler.com/bliki/DomainSpecificLanguage.html (Cited on page 70.)

[70] L. Franzotte and S. R. Vergilio, "Applying Mutation Testing to XML Schemas," in *SEKE*, 2006, pp. 511–516. (Cited on page 39.)

[71] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," Network Working Group, RfC 2045, 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2045.txt (Cited on page 99.)

[72] ——, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," Network Working Group, RfC 2046, 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2045.txt (Cited on page 99.)

[73] T. L. Friedman, *The world is flat: A brief history of the twenty-first century*. Macmillan, 2005. (Cited on page 2.)

[74] S. Gal-On and M. Levy, "Measuring Multicore Performance," *Computer*, vol. 41, no. 11, pp. 99–102, Nov. 2008. (Cited on page 49.)

[75] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Amsterdam: Addison-Wesley, 1995. (Cited on pages 168 and 169.)

[76] J. García-Fanjul and J. T. Claudio de la Riva, "Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking," in *Testing: Academic and Industrial Conference – Practice And Research Techniques*. Windsor, United Kingdom: IEEE, Aug. 2006. (Cited on page 189.)

[77] Gartner Inc., *Hype Cycle for Cloud Computing*, 2011. (Cited on page 154.)

[78] ——, *IT Glossary*, 2017. [Online]. Available: http://www.gartner.com/it-glossary (Cited on page 2.)

[79] M. Geiger and G. Wirtz, "BPMN 2.0 Serialization – Standard Compliance Issues and Evaluation of Modeling Tools," in *5th International Workshop on Enterprise Modelling and Information Systems Architectures*, St. Gallen, Switzerland, Sep. 2013. (Cited on pages 10, 109, 188, and 191.)

[80] M. Geiger, A. Schönberger, and G. Wirtz, "Towards Automated Conformance Checking of ebBP-ST Choreographies and Corresponding WS-BPEL Based Orchestrations," in *Conference on Software Engineering and Knowledge Engineering (SEKE), Miami, Florida, USA*. Knowledge Systems Institute, Jul. 2011. (Cited on page 189.)

[81] M. Geiger, S. Harrer, J. Lenhard, M. Casar, A. Vorndran, and G. Wirtz, "BPMN Conformance in Open Source Engines," in *Proceedings of the 9th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, IEEE. San Francisco Bay, CA, USA: Institute of Electrical & Electronics Engineers (IEEE), Mar. 2015, pp. 21–30. (Cited on pages xv, 16, 18, 47, 56, 58, 59, 64, 65, 66, 104, 116, 124, and 134.)

[82] M. Geiger, S. Harrer, and J. Lenhard, "Process Engine Benchmarking with Betsy in the Context of ISO/IEC Quality Standards," *Softwaretechnik-Trends*, vol. 36, no. 2, pp. 57–60, 2016. (Cited on page 18.)

[83] ——, "Process Engine Benchmarking with Betsy – Current Status and Future Directions," in *Proceedings of the 8th Central-European Workshop on Services and their Composition (ZEUS)*, Vienna, Austria, Jan. 2016, pp. 37–44. (Cited on page 18.)

[84] M. Geiger, S. Harrer, J. Lenhard, and G. Wirtz, "On the Evolution of BPMN 2.0 Support and Implementation," in *10th International IEEE Symposium on Service-Oriented System Engineering (SOSE)*. Oxford, UK: Institute of Electrical and Electronics Engineers (IEEE), Mar. 2016, pp. 120–128. (Cited on pages 18, 47, 64, 65, 66, 104, 107, and 175.)

[85] ——, "BPMN 2.0: The State of Support and Implementation," *Future Generation Computer Systems*, 2017. (Cited on pages xvi, 18, 64, 65, 109, 128, 130, and 131.)

[86] M. Geiger, P. Neugebauer, and A. Vorndran, "Automatic Standard Compliance Assessment of BPMN 2.0 Process Models," in *9th Central European Workshop on Services and their Composition*, Lugano, Switzerland, 2017, in press. (Cited on pages 76, 109, and 173.)

[87] M. Glinz, *A Glossary of Requirements Engineering Terminology. Version 1.6*, May 2014. (Cited on pages 11 and 68.)

[88] A. Grandori, "A Prescriptive Contingency View of Organizational Decision Making," *Administrative Science Quarterly*, vol. 29, no. 2, p. 192, Jun. 1984. (Cited on page 10.)

[89] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro, "On the Interplay Between Fault Handling and Request-Response Service Interactions,"

in *8th International Conference on Application of Concurrency to System Design*, Xi'an, China, Jun. 2008, pp. 190–198. (Cited on page 86.)

[90] C. Gutschier, R. Hoch, H. Kaindl, and R. Popp, "A Pitfall with BPMN Execution," in *Second International Conference on Building and Exploring Web Based Environments*, Chamonix, France, Apr. 2014, pp. 7–13. (Cited on page 5.)

[91] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, p. 494. (Cited on page 44.)

[92] P. Harmon and C. Wolf, *State of Business Process Management*, Mar. 2016. (Cited on page 5.)

[93] S. Harrer, "Process Engine Selection Support," in *On the Move to Meaningful Internet Systems: OTM 2014 Workshops*. Springer Berlin Heidelberg, Oct. 2014, vol. 8842, pp. 18–22. (Cited on page 18.)

[94] S. Harrer and J. Lenhard, "Betsy – A BPEL Engine Test System," Otto-Friedrich Universität Bamberg, Tech. Rep. 90, Jul. 2012. (Cited on pages xv, 16, 18, 47, 48, 56, 57, 60, 66, 116, and 152.)

[95] S. Harrer, J. Lenhard, and G. Wirtz, "BPEL Conformance in Open Source Engines," in *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE. Taipei, Taiwan: Institute of Electrical & Electronics Engineers (IEEE), Dec. 2012, pp. 237–244. (Cited on pages 18, 84, 116, 124, 128, 129, 134, 153, and 168.)

[96] ——, "Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines," in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC)*, ser. Lecture Notes in Computer Science, vol. 8274. Berlin, Germany: Springer Berlin Heidelberg, Dec. 2013, pp. 99–113. (Cited on pages xvi, 16, 18, 61, 66, 89, 124, 128, and 129.)

[97] S. Harrer, J. Lenhard, G. Wirtz, and T. van Lessen, "Towards Uniform BPEL Engine Management in the Cloud," in *Proceedings des CloudCycle14 Workshops auf der 44. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, ser. LNI. Gesellschaft für Informatik e.V. (GI), Sep. 2014. (Cited on pages xiii, 17, 18, 47, 48, 51, and 177.)

[98] S. Harrer, F. Nizamic, G. Wirtz, and A. Lazovik, "Towards a Robustness Evaluation Framework for BPEL Engines," in *Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. Matsue, Japan: IEEE, Nov. 2014, pp. 199–206. (Cited on pages xvi, 16, 18, 66, 96, 128, 129, 130, and 173.)

[99] S. Harrer, C. Preißinger, and G. Wirtz, "BPEL Conformance in Open Source Engines: The Case of Static Analysis," in *Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. Matsue, Japan: IEEE, Nov. 2014, pp. 33–40.

(Cited on pages xvi, 18, 66, 91, 94, 128, 129, and 173.)

[100] S. Harrer, C. Röck, and G. Wirtz, "Automated and Isolated Tests for Complex Middleware Products: The Case of BPEL Engines," in *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Cleveland, Ohio, USA, Apr. 2014, pp. 390–398, testing Tools Track. (Cited on pages 16, 17, 18, 47, 152, and 174.)

[101] S. Harrer, M. Geiger, C. R. Preißinger, D. Bimamisa, S. J. A. Schuberth, and G. Wirtz, "Improving the Static Analysis Conformance of BPEL Engines with BPELlint," in *Proceedings of the 9th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, IEEE. San Francisco Bay, CA, USA: Institute of Electrical & Electronics Engineers (IEEE), Mar. 2015. (Cited on pages 17, 18, 76, and 173.)

[102] S. Harrer, O. Kopp, and J. Lenhard, "Patterns for Workflow Engine Benchmarking," in *PATTWORLD*, 2016. (Cited on pages 17, 18, 168, and 171.)

[103] S. Harrer, J. Lenhard, O. Kopp, V. Ferme, , and C. Pautasso, "A Pattern Language for Workflow Engine Conformance and Performance Benchmarking," in *EuroPLoP*, 2017, (accepted). (Cited on pages 17, 171, and 190.)

[104] R. Harris, "Introduction to Decision Making," 2012, accessed 30-January-2017. [Online]. Available: http://www.virtualsalt.com/crebook5.htm (Cited on page 21.)

[105] F. Haupt, F. Leymann, A. Nowak, and S. Wagner, "Lego4TOSCA: Composable Building Blocks for Cloud Applications," in *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD)*. Anchorage, Alaska, USA: IEEE, 2014. (Cited on page 50.)

[106] M. Hertis and M. B. Juric, "An Empirical Analysis of Business Process Execution Language Usage," *IEEE Transactions on Software Engineering*, vol. 40, no. 8, pp. 738–757, May 2014. (Cited on page 98.)

[107] F. Herzberg, *One more time: How do you motivate employees*. Harvard Business Review Boston, MA, 1968. (Cited on page 7.)

[108] A. R. Hevner, M. T. Salvatore, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS quarterly*, vol. 28, no. 1, pp. 75–105, 2004. (Cited on pages xv, 15, and 183.)

[109] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *International Joint Conference on Artificial intelligence*, ser. IJCAI'73, San Francisco, CA, USA, 1973, pp. 235–245. (Cited on page 57.)

[110] E. Højsgaard and T. Hallwyl, "Core BPEL: syntactic simplification of WS-BPEL 2.0," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1984–1991. (Cited on pages 27 and 86.)

[111] K. Huppler, "The Art of Building a Good Benchmark," in *Performance Evaluation and Benchmarking*. Springer, 2009, pp. 18–30. (Cited on pages xv, 35, 36, 37, 66, 131, 186, and 188.)

[112] IETF, *Key words for use in RFCs to Indicate Requirement Levels*, Mar. 1997, rFC 2119. (Cited on page 84.)

[113] ISO/IEC, *ISO/IEC 25010:2011; Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, 2011. (Cited on pages xiii, xv, 7, 8, 32, 33, 34, 70, 78, 110, 115, and 189.)

[114] ——, *ISO/IEC 25041:2012; Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Evaluation guide for developers, acquirers and independent evaluators*, 2012. (Cited on pages xv, 6, 12, 32, and 34.)

[115] ——, *ISO/IEC 19510:2013 – Information technology – Object Management Group Business Process Model and Notation*, Nov. 2013, v2.0.2. (Cited on pages xiii, 4, 5, 25, 28, 29, 47, 104, 105, 106, and 119.)

[116] ——, *ISO/IEC 25000:2014; Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)*, 2014. (Cited on pages xv and 32.)

[117] ——, *ISO/IEC 25051:2014; Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Requirements for quality of Ready to Use Software Product (RUSP) and instructions for testing*, 2014. (Cited on pages xv, 3, 4, 8, 10, 12, 32, 34, and 68.)

[118] ISO/IEC/IEEE, *ISO/IEC/IEEE 24765:2010; Systems and software engineering – Vocabulary*, 2014. (Cited on pages xiv, 113, 114, and 136.)

[119] Java-Source.net, "Open Source Workflow Engines in Java," 2016, accessed 30-January-2017. [Online]. Available: http://java-source.net/open-source/workflow-engines (Cited on page 5.)

[120] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011. (Cited on page 39.)

[121] K. Jin and E. L. Miller, "The Effectiveness of Deduplication on Virtual Machine Disk Images," in *Proceedings of SYSTOR: The Israeli Experimental Systems Conference*. ACM, 2009, p. 7. (Cited on pages 41 and 44.)

[122] T. Jones, W. R. Schulte, and M. Cantara, "Magic Quadrant for Intelligent Business Process Management Suites," Gartner Research, Tech. Rep., 2014. (Cited on pages 8 and 9.)

[123] L. Juszczyk, H.-L. Truong, and S. Dustdar, "GENESIS – A Framework for Automatic Generation and Steering of Testbeds of Complex Web Services," in *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Mar. 2008, pp. 131–140. (Cited on pages 67, 118, and 188.)

[124] L. Juszczyk and S. Dustdar, "Script-Based Generation of Dynamic Testbeds for SOA," in *Socially Enhanced Services Computing*, S. Dustdar, D. Schall, F. Skopik, L. Juszczyk, and H. Psaier, Eds. Springer Vienna, 2011, pp. 77–94. (Cited on page 67.)

[125] D. Karagiannis, S. Junginger, and R. Strobl, "Introduction to Business Process Management Systems Concepts," in *Business Process Modelling*, B. Scholz-Reiter and E. Stickel, Eds. Springer Berlin Heidelberg, 1996, pp. 81–106. (Cited on page 3.)

[126] R. Khalaf, A. Keller, and F. Leymann, "Business processes for Web Services: Principles and applications," *IBM Systems Journal*, vol. 45, no. 2, pp. 425–446, 2006. (Cited on page 47.)

[127] R. Kimball, *The data warehouse lifecycle toolkit: expert methods for designing, developing, and deploying data warehouses*. John Wiley & Sons, 1998. (Cited on page 141.)

[128] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230. (Cited on page 42.)

[129] R. K. L. Ko, S. S. G. Lee, and E. Wah Lee, "Business process management (BPM) standards: a survey," *Business Process Management Journal*, vol. 15, no. 5, pp. 744–791, 2009. (Cited on page 5.)

[130] C. Kohls, "The structure of patterns," in *Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP)*. Association for Computing Machinery (ACM), 2010. (Cited on page 169.)

[131] ——, "The structure of patterns – Part II – Qualities," in *Proceedings of the 18th Conference on Pattern Languages of Programs (PLOP)*. ACM, 2011, p. 27. (Cited on page 169.)

[132] S. Kolb and C. Röck, "Unified Cloud Application Management," in *Proceedings of the 12th IEEE World Congress on Services*. San Francisco, CA, USA: IEEE, 2016. (Cited on page 49.)

[133] S. Kolb and G. Wirtz, "Towards Application Portability in Platform as a Service," in *Proceedings of the 8th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*. Oxford, United Kingdom: IEEE, Apr. 2014. (Cited on pages 49, 137, and 189.)

[134] O. Kopp, R. Mietzner, and F. Leymann, "Abstract Syntax of WS-BPEL 2.0," Universität Stuttgart, Tech. Rep. 6, Oct. 2008. (Cited on page 93.)

[135] O. Kopp, D. Martin, D. Wutke, and F. Leymann, "The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages," *Enterprise Modelling and Information Systems Architectures*, vol. 4, no. 1, pp. 3–13, 2009. (Cited on page 26.)

[136] O. Kopp, F. Leymann, and D. Wutke, "Fault Handling in the Web Service Stack," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds. Springer Berlin Heidelberg, 2010, vol. 6470, pp. 303–317. (Cited on page 99.)

[137] O. Kopp, K. Görlach, D. Karastoyanova, F. Leymann, M. Reiter, D. Schumm, M. Sonntag, S. Strauch, T. Unger, M. Wieland, and R. Khalaf, "A Classification of BPEL Extensions," *Journal of Systems Integration*, vol. 2, no. 4, pp. 3–28, Nov. 2011. (Cited on page 28.)

[138] O. Kopp, S. Henke, D. Karastoyanova, R. Khalaf, F. Leymann, M. Sonntag, T. Steinmetz, T. Unger, and B. Wetzstein, "An Event Model for WS-BPEL 2.0," University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, Tech. Rep. 2011/07, Sep. 2011. (Cited on page 50.)

[139] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC)*, ser. LNCS, vol. 8274. Springer Berlin Heidelberg, Dec. 2013, Demonstration, pp. 700–704. (Cited on page 154.)

[140] H. Koziolek, "Performance Evaluation of Component-based Software Systems: A Survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010, special Issue on Software and Performance. (Cited on page 118.)

[141] R. Kuhn, Y. Lei, and R. Kacker, "Practical Combinatorial Testing: Beyond Pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, May 2008. (Cited on page 39.)

[142] V. S. Lai, R. P. Trueblood, and B. K. Wong, "Software selection: a case study of the application of the analytical hierarchical process to the selection of a multimedia authoring system," *Information and Management*, vol. 36, no. 4, pp. 221–232, 1999. (Cited on page 22.)

[143] V. S. Lai, B. K. Wong, and W. Cheung, "Group decision making in a multiple criteria environment: A case using the AHP in software selection," *European Journal of Operational Research*, vol. 137, no. 1, pp. 134–144, 2002. (Cited on page 22.)

[144] A. Lanz, B. Weber, and M. Reichert, "Time Patterns in Process-aware Information Systems: A Pattern-based Analysis – Revised version." University of Ulm, Germany, Tech. Rep., 2009. (Cited on page 30.)

[145] P. K. Lawlis, K. E. Mark, D. A. Thomas, and T. Courtheyn, "A Formal Process for Evaluating COTS Software Products," *Computer*, vol. 34, no. 5, pp. 58–63, 2001. (Cited on pages 6, 10, 11, and 34.)

[146] S. C. Lee and J. Offutt, "Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis," in *ISSRE*, 2001. (Cited on page 39.)

[147] J. Lenhard, "A Pattern-based Analysis of WS-BPEL and Windows Workflow," Otto-Friedrich Universität Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, no. 88, Mar. 2011. (Cited on page 90.)

[148] ——, "Portability of Process-Aware and Service-Oriented Software: Evidence and Metrics," Ph.D. dissertation, University of Bamberg, 2016.

(Cited on pages 47 and 189.)

[149] J. Lenhard, A. Schönberger, and G. Wirtz, "Edit Distance-Based Pattern Support Assessment of Orchestration Languages," in *On the Move Confederated International Conferences: CoopIS, IS, DOA and ODBASE,*, Hersonissos, 2011. (Cited on pages 30, 89, and 90.)

[150] J. Lenhard, S. Harrer, and G. Wirtz, "Measuring the Installability of Service Orchestrations Using the SQuaRE Method," in *Proceedings of the 6th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. Kauai, Hawaii, USA: IEEE, Dec. 2013. (Cited on pages xv, 13, 14, 18, 47, 48, 60, 65, and 153.)

[151] J. Lenhard, V. Ferme, S. Harrer, M. Geiger, and C. Pautasso, "Lessons Learned from Evaluating Workflow Management Systems," in *ICSOC*, 2017, (under review). (Cited on pages 17 and 190.)

[152] T. v. Lessen, D. Lübke, and J. Nitzsche, *Geschäftsprozesse automatisieren mit BPEL [Automating Business Processes with BPEL]*. dpunkt.verlag, 2011. (Cited on page 9.)

[153] F. Leymann, "BPEL vs. BPMN 2.0: Should You Care?" in *Business Process Modeling Notation*, ser. Lecture Notes in Business Information Processing, J. Mendling, M. Weidlich, and M. Weske, Eds. Springer Berlin Heidelberg, 2010, vol. 67, pp. 8–13. (Cited on page 8.)

[154] F. Leymann and D. Roller, *Production Workflow – Concepts and Techniques*. Prentice Hall PTR, 2000. (Cited on page 3.)

[155] F. Leymann, D. Karastoyanova, and M. Papazoglou, *Handbook on Business Process Management*. Springer, 2010, ch. Business Process Management Standards. (Cited on page 25.)

[156] A. Lindsay, D. Downs, and I. E. C. Ken Lunn, "Business processes—attempts to find a definition," *Information and Software Technology*, vol. 45, no. 15, pp. 1015–1019, Dec. 2003. (Cited on page 2.)

[157] R. Lipshitz and O. Strauss, "Coping with Uncertainty: A Naturalistic Decision-Making Analysis," *Organizational Behavior and Human Decision Processes*, vol. 69, no. 2, pp. 149–163, Feb. 1997. (Cited on pages 7 and 10.)

[158] R. Lu and S. Sadiq, "A Survey of Comparative Business Process Modeling Approaches," in *Proceedings of the 10th International Conference on Business Information Systems (BIS), number 4439 in LNCS*. Springer, 2007, pp. 82–94. (Cited on page 189.)

[159] A. Lübbe and S. Schnägelberger, "BPM Toolmarktmonitor [BPM tool market survey]," BPM&O, Tech. Rep., 2015. (Cited on pages 9 and 11.)

[160] ——, "BPM Toolmarktmonitor Anwenderumfrage [BPM tool market user survey]," BPM&O, Tech. Rep., 2015. (Cited on pages 9 and 11.)

[161] D. Lübke, "Unit Testing BPEL Compositions," in *Test and Analysis of Service-oriented Systems*, L. Baresi and E. D. Nitto, Eds. Springer, 2007, pp. 149–171, iSBN 978-3-540-72911-2. (Cited on pages 67, 118, 185,

and 187.)

[162] T. Lynn, N. O'Carroll, J. Mooney, M. Helfert, D. Corcoran, G. Hunt, L. Van Der Werff, J. Morrison, and P. Healy, "Towards a framework for defining and categorising business Process-As-A-Service (BPaaS)," in *Proceedings of the 21st International Product Development Management Conference*, 2014. (Cited on page 65.)

[163] M. Makki, D. V. Landuyt, and W. Joosen, "Automated regression testing of BPMN 2.0 processes: a capture and replay framework for continuous delivery," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences - GPCE 2016*. Association for Computing Machinery (ACM), 2016. (Cited on page 118.)

[164] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decision support systems*, vol. 15, no. 4, pp. 251–266, 1995. (Cited on page 15.)

[165] B. D. Martino, G. Cretella, and A. Esposito, "Cross-Platform Cloud APIs," in *Cloud Portability and Interoperability*. Springer, 2015, pp. 45–57. (Cited on page 49.)

[166] A. P. Mathur, *Foundations of Software Testing*. Dorling Kindersley, 2009. (Cited on page 87.)

[167] S. Mathur and S. Malik, "Advancements in the V-Model," *International Journal of Computer Applications*, vol. 1, no. 12, pp. 30–35, Feb. 2010. (Cited on page 69.)

[168] P. Mayer and D. Lübke, "Towards a BPEL unit testing framework," in *Proceedings of the Workshop on Testing, analysis, and verification of web services and applications - TAV-WEB*. ACM, 2006. (Cited on pages 67, 118, 185, and 187.)

[169] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," NIST, NIST Special Publication 800-145, Sep. 2011. (Cited on page 40.)

[170] G. Meszaros and J. Doble, "A pattern language for pattern writing," *Pattern languages of program design*, vol. 3, pp. 529–574, 1998. (Cited on page 169.)

[171] H. Mili, G. Tremblay, G. B. Jaoude, E. Lefebvre, L. Elabed, and G. E. Boussaidi, "Business Process Modeling Languages: Sorting Through the Alphabet Soup," *ACM Computing Surveys*, vol. 43, no. 1, pp. 41–456, Dec. 2010. (Cited on pages 4, 5, and 25.)

[172] D. W. Miller and M. K. Starr, *Executive decisions and operations research*. Englewood Cliffs, 1963. (Cited on page 22.)

[173] P. E. Moody, *Decision making: Proven methods for better decisions*. McGraw-Hill Companies, 1983. (Cited on page 21.)

[174] N. Mulyar, W. M. P. van der Aalst, L. Aldred, and N. Russell, "Service Interaction Patterns: A Configurable Framework," in *BPM Center Report BPM-07-07*, 2007. (Cited on page 30.)

[175] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011. (Cited on pages 37 and 39.)

[176] G. Nikol, M. Träger, S. Harrer, and G. Wirtz, "Service-oriented Multi-tenancy (SO-MT): Enabling Multi-tenancy for Existing Service Composition Engines with Docker," in *10th International IEEE Symposium on Service-Oriented System Engineering (SOSE)*. Oxford, UK: IEEE, Mar. 2016, pp. 238–243. (Cited on pages 18 and 190.)

[177] OASIS, *Web Services Base Notification*, Oct. 2006, version 1.3. (Cited on page 50.)

[178] ——, *Web Services Brokered Notification*, Oct. 2006, version 1.3. (Not cited.)

[179] ——, *Web Services Topics*, Oct. 2006, version 1.3. (Cited on page 50.)

[180] ——, *Web Services Security*, Feb. 2006, v1.1. (Cited on page 28.)

[181] ——, *Web Services Business Process Execution Language*, Apr. 2007, v2.0. (Cited on pages 4, 5, 25, 26, 27, 47, 84, 85, 91, and 98.)

[182] ——, *Web Services Reliable Messaging*, Feb. 2009, v1.2. (Cited on page 28.)

[183] ——, *Cloud Application Management for Platforms*, Jul. 2013, version 1.1 – Draft 03. (Cited on page 49.)

[184] ——, *Topology and Orchestration Specification for Cloud Applications*, Nov. 2013, v1.0. (Cited on pages 50, 154, and 166.)

[185] OCCI, *Open Cloud Computing Interface – Core*, Open Grid Forum, 2011. (Cited on page 49.)

[186] OED, "Oxford English Dictionary Online," Mar. 2015, accessed 30-January-2017. [Online]. Available: http://www.oed.com/ (Cited on page 35.)

[187] J. Offutt, "A Practical System for Mutation Testing: Help for the Common Programmer," in *ITC*, 1994. (Cited on page 39.)

[188] OMG, *Business Process Model and Notation*, Jan. 2009, v1.2. (Cited on page 25.)

[189] ——, *Business Process Model and Notation*, Jan. 2011, v2.0. (Cited on pages 4 and 25.)

[190] ——, "BPMN Implementers," 2016, accessed 30-January-2017. [Online]. Available: http://www.bpmn.org/#tabs-implementers (Cited on page 5.)

[191] ——, "BPM Vendor Directory Listing," 2016, accessed 30-January-2017. [Online]. Available: http://bpm-directory.omg.org/vendor/list.htm (Cited on page 5.)

[192] D. Oppenheimer, A. B. Brown, J. Traupman, P. Broadwell, and D. A. Patterson, "Practical Issues in Dependability Benchmarking," in *Evaluating and Architecting System Dependability (EASY'02)*, 2002, p. 7. (Cited on page 134.)

[193] Oracle, "Quick Start Guide for Oracle® SOA Suite 11gR1 (11.1.1.5.0)," May 2012, accessed 30-January-2017. [Online]. Available: http://www.oracle.com/technetwork/middleware/soasuite/overview/quickstartguidesoasuite11gr1ps4-459545.pdf (Cited on pages 14 and 152.)

[194] OW2 Consortium, *Orchestra User Guide*, Oct. 2011. (Cited on page 57.)

[195] C. Pautasso, V. Ferme, D. Roller, F. Leymann, and M. Skouradaki, "Towards Workflow Benchmarking: Open Research Challenges," in *Proceedings of the 16th Conference on Database Systems for Business, Technology, and Web*, Hamburg, Germany, Mar. 2015, pp. 1–20. (Cited on page 185.)

[196] K. Pauwels, T. Ambler, B. H. Clark, P. LaPointe, D. Reibstein, B. Skiera, B. Wierenga, and T. Wiesel, "Dashboards As a Service: Why, What, How, and What Research is Needed?" *Journal of Service Research*, 2009. (Cited on pages 135 and 136.)

[197] C. Peltz, "Web Services Orchestration and Choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct. 2003. (Cited on page 2.)

[198] D. Petrova-Antonova, S. Ilieva, and D. Manova, "TASSA: Testing Framework for Web Service Orchestrations," in *IEEE/ACM 10th International Workshop on Automation of Software Test (AST)*, May 2015, pp. 8–12. (Cited on page 188.)

[199] A. Polyvyanyy, S. Smirnov, and M. Weske, "Process Model Abstraction: A Slider Approach," in *12th International IEEE on Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 2008, pp. 325–331. (Cited on page 119.)

[200] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974. (Cited on pages 40 and 41.)

[201] C. R. Preißinger, S. Harrer, S. J. A. Schuberth, D. Bimamisa, and G. Wirtz, "Towards Standard Conformant BPEL Engines: The Case of Static Analysis," in *Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS)*, Potsdam, Germany, Feb. 2014, pp. 60–63. (Cited on pages 18 and 95.)

[202] C. R. Preißinger and S. Harrer, "Static Analysis Rules of the BPEL Specification: Tagging, Formalization and Tests," Otto-Friedrich Universität Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik no. 99, Aug. 2014. (Cited on pages xvi, 18, 91, 93, and 194.)

[203] J. Radatz *et al.*, *IEEE Standard Glossary of Software Engineering Terminology*, Dec. 1990, IEEE Std 610.12-1990. (Cited on pages 11, 12, 35, 37, 38, 40, and 68.)

[204] T. Rademakers, *Activiti in Action: Executable Business Processes in BPMN 2.0*. Greenwich, CT, USA: Manning Publications Co., 2012. (Cited on page 9.)

[205] M. Rahman, Z. Chen, and J. Gao, "A Service Framework for Parallel Test Execution on a Developer's Local Development Workstation," in *Proceedings of the 9th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*. San Francisco Bay, CA, USA: IEEE, Mar. 2015. (Cited on pages 43, 44, and 155.)

[206] H. A. Reijers, "Implementing BPM systems: the role of process orientation," *Business Process Management Journal*, vol. 12, no. 4, pp. 389–409, Jul. 2006. (Cited on page 3.)

[207] M. D. Resnik, *Choices: An Introduction to Decision Theory*. University of Minnesota Press, 1987. (Cited on pages 20 and 21.)

[208] C. Richardson, C. L. Clair, A. Cullen, and S. McGovern, "Prepare For 2015's Shifting BPM Landscape," Forrester Research, Tech. Rep., Feb. 2015. (Cited on page 5.)

[209] L. Rising and N. S. Janoff, "The Scrum software development process for small teams," *IEEE Software*, vol. 17, no. 4, p. 26, 2000. (Cited on page 138.)

[210] C. Röck and S. Harrer, "Testing BPEL Engine Performance: A Survey," Otto-Friedrich Universität Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik no. 93, May 2014. (Cited on page 18.)

[211] C. Röck and S. Kolb, "Nucleus – Unified Deployment and Management for Platform as a Service," University of Bamberg, Tech. Rep., 2016. (Cited on page 49.)

[212] C. Röck, S. Harrer, and G. Wirtz, "Performance Benchmarking of BPEL Engines: A Comparison Framework, Status Quo Evaluation and Challenges," in *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Vancouver, Canada, Jul. 2014, pp. 31–34. (Cited on page 18.)

[213] D. Roller, "Throughput Improvements for BPEL Engines: Implementation Techniques and Measurements Applied to SWoM," Ph.D. dissertation, Institute of Architecture of Application Systems (IAAS), Stuttgart, Germany, Jul. 2013. (Cited on page 118.)

[214] P. Rook, "Controlling software projects," *Software Engineering Journal*, vol. 1, no. 1, p. 7, 1986. (Cited on page 69.)

[215] G. Rosinosky, S. Youcef, and F. Charoy, "A Framework for BPMS Performance and Cost Evaluation on the Cloud," in *Workshop*" *Business Process Monitoring and Performance Analysis in the Cloud*", 2016. (Cited on page 118.)

[216] B. Roy, "The outranking approach and the foundations of ELECTRE methods," *Theory and decision*, vol. 31, no. 1, pp. 49–73, 1991. (Cited on page 22.)

[217] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Workflow Data Patterns," Queensland University of Technology,

Brisbane, Australia, Tech. Rep., 2004. (Cited on page 30.)

[218] ——, "Workflow Resource Patterns," Eindhoven University of Technology, BETA Working Paper Series, Tech. Rep., 2004. (Cited on page 30.)

[219] N. Russell, A. H. M. ter Hofstede, and D. Edmond, "Workflow Resource Patterns: Identification, Representation and Tool Support," in *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE), volume 3520 of Lecture Notes in Computer Science*. Porto, Portugal: Springer, Jun. 2005, pp. 216–232. (Cited on page 30.)

[220] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Workflow Data Patterns: Identification, Representation and Tool Support," in *Proceedings of the 24th International Conference on Conceptual Modeling (ER)*. Klagenfurt, Austria: Springer, Oct. 2005, pp. 353–368. (Cited on page 30.)

[221] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede, "Workflow Exception Patterns," in *Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE)*, Luxembourg, Luxembourg, Jun. 2006, pp. 288–302. (Cited on page 30.)

[222] T. L. Saaty, "How to make a decision: The Analytic Hierarchy Process," *European Journal of Operational Research*, vol. 48, no. 1, pp. 9–26, 1990. (Cited on pages 6, 20, 22, and 23.)

[223] ——, "Analytic Network Process," in *Encyclopedia of Operations Research and Management Science*. Springer, 2001, pp. 28–35. (Cited on page 22.)

[224] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann, "Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark," *Performance Evaluation*, vol. 66, no. 8, pp. 410–434, 2009. (Cited on page 118.)

[225] J. Savolainen, J. Kuusela, and A. Vilavaara, "Transition to agile development-rediscovery of important requirements engineering practices," in *18th IEEE International Requirements Engineering Conference*. IEEE, 2010, pp. 289–294. (Cited on page 138.)

[226] G. Scheithauer and P. Klinnert, "Optimierung und Standardisierung von Leasing & Retail Finance Prozessen bei einer internationalen Automobilbank," Capgemini Consulting, Tech. Rep., 2015. (Cited on pages 8 and 9.)

[227] S. Schlauderer and S. Overhage, "Selecting Cloud Service Providers-Towards a Framework of Assessment Criteria and Requirements," in *Proceedings der 12. Internationalen Tagung Wirtschaftsinformatik (WI)*, Osnabrück, Germany, 2015, pp. 76–90. (Cited on page 189.)

[228] D. Schumm, F. Leymann, and A. Streule, "Process viewing patterns," in *14th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 2010, pp. 89–98. (Cited on pages 30 and 119.)

[229] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using Benchmarking to Advance Research: A Challenge to Software Engineering," in *Proceedings*

of the 25th International Conference on Software Engineering. IEEE Computer Society, 2003, pp. 74–83. (Cited on pages xv, 36, 37, 66, 131, and 186.)

[230] J. Sinur and J. B. Hill, "Magic Quadrant for Business Process Management Suites," Gartner Research, Tech. Rep., 2010. (Cited on pages 8 and 9.)

[231] J. Sinur, W. R. Schulte, J. B. Hill, and T. Jones, "Magic Quadrant for Intelligent Business Process Management Suites," Gartner Research, Tech. Rep., 2012. (Cited on pages 8 and 9.)

[232] M. Skouradaki, D. H. Roller, F. Leymann, V. Ferme, C. Pautasso *et al.*, "Technical Open Challenges on Benchmarking Workflow Management Systems," in *Technical Report of the Symposum on Software Performance (SOSP)*. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2014, pp. 1–8. (Cited on page 185.)

[233] M. Skouradaki, K. Goerlach, M. Hahn, and F. Leymann, "Application of Sub-Graph Isomorphism to Extract Reoccurring Structures from BPMN 2.0 Process Models," in *2015 IEEE Symposium on Service-Oriented System Engineering*. Institute of Electrical and Electronics Engineers (IEEE), Mar. 2015. (Cited on page 185.)

[234] M. Skouradaki, D. H. Roller, F. Leymann, V. Ferme, and C. Pautasso, "On the Road to Benchmarking BPMN 2.0 Workflow Engines," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 301–304. (Cited on pages 172 and 185.)

[235] M. Skouradaki, V. Andrikopoulos, and F. Leymann, "Representative BPMN 2.0 Process Model Generation from Recurring Structures," in *2016 IEEE International Conference on Web Services (ICWS)*. Institute of Electrical and Electronics Engineers (IEEE), Jun. 2016. (Cited on pages 185 and 186.)

[236] M. Skouradaki, T. Azad, U. Breitenbuecher, O. Kopp, and F. Leymann, "A Decision Support System for the Performance Benchmarking of Workflow Management Systems," in *Proceedings of the 10th Symposium and Summer School On ServiceOriented Computing, SummerSOC 2016*. IBM, Sep. 2016, pp. 41–57. (Cited on pages 67, 111, 137, 185, and 186.)

[237] M. Skouradaki, V. Ferme, C. Pautasso, F. Leymann, and A. van Hoorn, "Micro-Benchmarking BPMN 2.0 Workflow Management Systems with Workflow Patterns," in *28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016, Proceedings,Springer Lecture Notes in Computer Science*. Springer, Jun. 2016, pp. 67–82. (Cited on pages 5, 8, 9, 10, 110, 111, 115, 142, 172, 185, and 186.)

[238] A. J. Smith, "Workloads (Creation and Use)," *Communications of the ACM*, vol. 50, no. 11, pp. 45–50, Nov. 2007. (Cited on page 170.)

[239] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005. (Cited on page 41.)

[240] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. (Cited on page 41.)

[241] M. Sonntag and D. Karastoyanova, "Ad hoc Iteration and Re-execution of Activities in Workflows," *International Journal On Advances in Software*, vol. 5, no. 1 & 2, pp. 91–109, Jul. 2012. (Cited on page 55.)

[242] ——, "Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows," *Journal of Grid Computing*, vol. 11, no. 3, pp. 553–583, Sep. 2013. (Cited on page 50.)

[243] SPEC, *SPEC VIRT_SC*, Sep. 2013, v1.1. (Cited on page 35.)

[244] G. Stalk, P. Evans, and L. E. Shulman, "Competing on capabilities: The new rules of corporate strategy," *Harvard business review*, vol. 70, no. 2, pp. 57–69, 1991. (Cited on page 2.)

[245] S. Strauch, V. Andrikopoulos, S. G. Saéz, and F. Leymann, "Implementation and Evaluation of a Multi-tenant Open-Source ESB," in *Proceedings of the European Conference on Service-Oriented and Cloud Computing (ESOCC)*, ser. Lecture Notes in Computer Science (LNCS), vol. 8135. Springer Berlin Heidelberg, 2013, pp. 79–93. (Cited on page 118.)

[246] G. H. Subramanian, J. Nosek, S. P. Raghunathan, and S. S. Kanitkar, "A Comparison of the Decision Table and Tree," *Communications of the ACM*, vol. 35, no. 1, pp. 89–94, Jan. 1992. (Cited on page 21.)

[247] F. Tarawneh, F. Baharom, J. H. Yahaya, and F. Ahmad, "Evaluation and Selection COTS Software Process: The State of the Art," *International Journal of New Computer Architectures and their Applications (IJNCAA)*, vol. 1, no. 2, pp. 344–357, 2011. (Cited on pages 6, 11, and 34.)

[248] TEC, "Business Process Management (BPM) Software Evaluation Report," Online, accessed 30-January-2017. [Online]. Available: https://www3.technologyevaluation.com/store/software-evaluation-report/business-process-management-bpm-software-evaluation-report.html (Cited on pages 8 and 10.)

[249] ——, "Business Process Management (BPM) RFI/RFP Template," Online, accessed 30-January-2017. [Online]. Available: https://www3.technologyevaluation.com/store/rfp-template/business-process-management-bpm-rfi-rfp-template.html (Cited on page 10.)

[250] L. H. Thom, M. Reichert, and C. Iochpe, "Activity Patterns in Process-aware Information Systems: Basic Concepts and Empirical Evidence," *International Journal of Business Process Integration and Management*, vol. 4, no. 2, pp. 93–110, 2009. (Cited on page 30.)

[251] S. Thöne, R. Depke, and G. Engels, "Process-Oriented, Flexible Composition of Web Services with UML," in *Advanced Conceptual Modeling Techniques*. Springer, 2003, pp. 390–401. (Cited on page 189.)

[252] J. Thones, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015. (Cited on page 190.)

[253] TPC, *TPC Virtual Measurement Single System*, Nov. 2013, v1.2.0. (Cited on page 35.)

[254] E. Triantaphyllou, *Multi-criteria decision making methods: a comparative study*. Springer Science & Business Media, 2013, vol. 44. (Cited on pages 6, 21, and 22.)

[255] A. M. Turing, "Systems of Logic Based on Ordinals: a Dissertation," Ph.D. dissertation, Cambridge University, Cambridge, UK, 1938. (Cited on page 89.)

[256] O. S. Vaidya and S. Kumar, "Analytic hierarchy process: An overview of applications," *European Journal of Operational Research*, vol. 169, no. 1, pp. 1–29, 2006. (Cited on page 22.)

[257] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, Jul. 2003. (Cited on pages 30, 31, 80, 90, 107, 108, 172, and 186.)

[258] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek, "Conformance Checking of Service Behavior," *ACM Transactions on Internet Technology*, vol. 8, no. 3, pp. 1–29, May 2008. (Cited on page 189.)

[259] W. M. P. van der Aalst, A. Mooij, C. Stahl, and K. Wolf, "Service Interaction: Patterns, Formalization, and Analysis," in *Formal Methods for Web Services*, ser. LNCS, vol. 5569. Springer Berlin/ Heidelberg, 2009, pp. 42–88, iSBN: 978-3-642-01917-3. (Cited on page 30.)

[260] W. M. P. van der Aalst, "Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management," in *Lectures on Concurrency and Petri Nets*. Springer Nature, 2004, pp. 1–65. (Cited on page 24.)

[261] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske, "Business Process Management: A Survey," in *Proceedings of the International Conference on Business Process Management*. Eindhoven, The Netherlands: Springer Berlin Heidelberg, Jun. 2003, pp. 1–12. (Cited on pages xiii, 2, 24, 47, and 48.)

[262] W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf, "From Public Views to Private Views – Correctness-by-Design for Services," in *4th International Workshop on Web Services and Formal Methods (WS-FM)*, Sep. 2007, pp. 139–153. (Cited on page 189.)

[263] A. Van Deursen, P. Klint, J. Visser *et al.*, "Domain-specific languages: An annotated bibliography," *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000. (Cited on page 25.)

[264] A. van Dijk, "Contracting Workflows and Protocol Patterns," in *International Conference on Business Process Management*, Eindhoven, The

Netherlands, Jun. 2003, pp. 152–167. (Cited on page 30.)

[265] T. van Lessen, F. Leymann, R. Mietzner, J. Nitzsche, and D. Schleicher, "A Management Framework for WS-BPEL," in *Proceedings of the 6th IEEE European Conference on Web Services*. Dublin, Ireland: IEEE Computer Society, Nov. 2008, Conference Paper, pp. 187–196. (Cited on page 50.)

[266] W. Vogels, "Beyond server consolidation," *Queue*, vol. 6, no. 1, pp. 20–26, 2008. (Cited on page 40.)

[267] J. Vom Brocke and M. Rosemann, *Handbook on Business Process Management 1 – Introduction, Methods, and Information Systems*, 2nd ed. Springer, 2010. (Cited on page 23.)

[268] K. Vukojevic-Haupt, F. Haupt, and F. Leymann, "On-demand provisioning of workflow middleware and services into the cloud: an overview," *Computing*, Oct. 2016. (Cited on page 51.)

[269] W3C, *XML Path Language (XPath) Version 1.0*, Nov. 1999, v1.0. (Cited on page 93.)

[270] ——, *Simple Object Access Protocol (SOAP) 1.1*, 2000. (Cited on pages 26 and 49.)

[271] ——, *Web Services Description Language (WSDL) 1.1*, Mar. 2001, v1.1. (Cited on pages 26 and 87.)

[272] ——, *Web Services Addressing 1.0 – Core*, May 2006. (Cited on page 85.)

[273] ——, *Extensible Markup Language (XML) Version 1.0*, 5th ed., Nov. 2008, v1.0. (Cited on page 100.)

[274] ——, *Namespaces in XML*, Dec. 2009, v1.0. (Cited on page 52.)

[275] B. Weber, S. Rinderle, and M. Reichert, "Change Support in Process-Aware Information Systems – A Pattern-Based Analysis," University of Twente, The Netherlands, Tech. Rep., 2007. (Cited on page 30.)

[276] ——, "Change Patterns and Change Support Features – Enhancing Flexibility in Process-Aware Information Systems," *Data and Knowledge Engineering*, vol. 66, no. 3, pp. 438–466, Jul. 2008. (Cited on page 30.)

[277] C.-C. Wei, C.-F. Chien, and M.-J. J. Wang, "An AHP-based approach to ERP system selection," *International Journal of Production Economics*, vol. 96, no. 1, pp. 47–62, 2005. (Cited on page 22.)

[278] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer Science & Business Media, 2012. (Cited on pages xiii, 2, 3, 4, 23, 24, 25, and 190.)

[279] WfMC, *Terminology & Glossary*, WfMC, Feb. 1999, v3.0. (Cited on pages 3 and 24.)

[280] ——, *XML Process Definition Language*, Aug. 2012, v2.2. (Cited on page 25.)

[281] WfMC, "XPDL Implementations," 2016, accessed 30-January-2017. [Online]. Available: http://www.wfmc.org/53-standards/xpdl/89-xpdl-implementations (Cited on page 25.)

[282] Wikipedia, "List of BPEL engines — Wikipedia, The Free Encyclopedia," 2016, accessed 30-January-2017. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_BPEL_engines&oldid=748138912 (Cited on page 5.)

[283] ——, "List of BPMN 2.0 engines — Wikipedia, The Free Encyclopedia," 2016, accessed 30-January-2017. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_BPMN_2.0_engines&oldid=752482920 (Cited on page 5.)

[284] R. Winter, "Design science research in Europe," *European Journal of Information Systems*, vol. 17, no. 5, pp. 470–475, 2008. (Cited on page 15.)

[285] P. Wohed, W. M. P. v. d. Aalst, M. Dumas, A. H. M. t. Hofstede, and N. Russell, "Pattern-based Analysis of BPMN – An extensive evaluation of the Control-flow, the Data and the Resource Perspectives," BPM Center, BPMcenter.org, BPM Center Report BPM-05-26, 2005. (Cited on page 30.)

[286] P. Wohed, M. Dumas, A. H. M. T. Hofstede, and N. Russell, "Pattern-based Analysis of BPMN – An extensive evaluation of the Control-flow, the Data and the Resource Perspectives (revised version)," BPM Center, BPMcenter.org, BPM Center Report BPM-06-17, 2006. (Cited on page 172.)

[287] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russell, "On the Suitability of BPMN for Business Process Modelling," in *BPM*, 2006, pp. 161–176. (Cited on pages 107 and 108.)

[288] P. Wohed, N. Russell, A. H. M. Ter Hofstede, B. Andersson, and W. M. P. van der Aalst, "Patterns-based Evaluation of Open Source BPM Systems: The Cases of jBPM, OpenWFE, and Enhydra Shark," *Information and Software Technology*, vol. 51, no. 8, pp. 1187–1216, 2009. (Cited on page 30.)

[289] P. Wohed, A. H. M. Ter Hofstede, N. Russell, B. Andersson, and W. M. P. van der Aalst, "On the maturity of open source BPM systems," BPTrends, Tech. Rep., 2009. (Cited on pages 8, 9, 10, and 190.)

[290] R. Wohlstetter, *Pearl Harbor: warning and decision*. Stanford University Press, 1962. (Cited on page 10.)

[291] Workflow Patterns Initiative, "Evaluations," 2016, accessed 30-January-2017. [Online]. Available: http://www.workflowpatterns.com/evaluations/ (Cited on pages 8, 10, 30, and 137.)

[292] WS-I, *Basic Profile*, Nov. 2010, v2.0. [Online]. Available: http://ws-i.org/Profiles/BasicProfile-2.0-2010-11-09.html (Cited on page 87.)

[293] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance Evaluation of Container-based Virtualization for High Performance Computing Environments," in *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*

(PDP). IEEE, 2013, pp. 233–240. (Cited on pages 40, 41, 42, 155, and 189.)

[294] O. M. Yigitbasioglu and O. Velcu, "A Review of Dashboards in Performance Management: Implications for Design and Research," *IJAIS*, vol. 13, no. 1, pp. 41–59, 2012. (Cited on pages 134, 135, and 136.)

[295] H.-J. Zimmermann and L. Gutsche, *Multi-Criteria Analyse: Einführung in die Theorie der Entscheidungen bei Mehrfachzielsetzungen [Multi Criteria Analysis: Introduction into the Theory of Decisions with Multiple Objectives]*. Springer-Verlag, 2013. (Cited on page 21.)

University of Bamberg Press

Business processes have become ubiquitous in industry today. They form the main ingredient of business process management. The two most prominent standardized languages to model business processes are Web Services Business Process Execution Language 2.0 (BPEL) and Business Process Model and Notation 2.0 (BPMN). Business process engines allow for automatic execution of business processes. There is a plethora of business process engines available, and thus, one has the agony of choice: which process engine fits the demands the best? The lack of objective, reproducible, and ascertained information about the quality of such process engines makes rational choices very difficult.
This can lead to baseless and premature decisions that may result in higher long term costs. This work provides an effective and efficient benchmarking solution to reveal the necessary information to allow making rational decisions. The foundation comprises an abstraction layer for process engines that provides a uniform API to interact with any engine similarly and a benchmark language for process engines to represent benchmarks in a concise, self-contained, and interpretable domain-specific language. A benchmark framework for process engines performs benchmarks represented in this language on engines implementing the abstraction layer. The produced benchmark results are visualized and made available for decision makers via a public interactive dashboard. On top of that, the efficient benchmark framework uses virtual machines to improve test isolation and reduce "time to result" by snapshot restoration accepting a management overhead. Based on the gained experience, eight challenges faced in process engine benchmarking are identified, resulting in 21 process engine benchmarking. Results show that this approach is both effective and efficient. Effective because it covers four BPEL-based and another four BPMN-based benchmarks which cover half of the quality characteristics defined by the ISO/IEC 25010 product quality model. Efficient because it fully automates the benchmarking of process engines and can leverage virtualization for an even higher execution efficiency. With this approach, the barrier for creating good benchmarks is significantly lowered. This allows decision makers to consistently evaluate process engines and, thus, makes rational decisions for the corresponding selection possible.

eISBN: 978-3-86309-504-8

9 783863 095048

www.uni-bamberg.de/ubp