# Improving Process Portability through Metrics and Continuous Inspection

Jörg Lenhard

Distributed Systems Group, University of Bamberg, Germany
joerg.lenhard@uni-bamberg.de

**Abstract.** Runtimes for process-aware applications, i.e., process engines, constantly evolve and in the age of cloud-enabled process execution, the need to change a runtime quickly becomes even more evident. To cope with this fast pace, it is desirable to build processes in a way that makes them easily portable among engines. Reliance on process standards is a step in the right direction, but cannot solely solve all problems. Standards are just specifications from which implementations will naturally deviate, thus fueling the problem of process portability. Here, the field of software measurement can provide some remedy. Metrics for process portability can help to make intelligent decisions on whether to invest in porting or rewriting process-aware applications. What is more, if integrated into the development process through agile techniques like continuous inspection, portability metrics can help in the implementation of more portable processes from the very beginning.
In this chapter, we present an approach for the measurement of process portability and explain how this can improve decision making and process quality in general. The approach builds on the recently revised version of the renowned ISO/IEC software quality model and we describe how this model is in line with techniques of continuous inspection. We discuss what constitutes process portability and present a set of newly proposed software metrics for quantifying portability.

**Keywords:** Portability, Process Quality, Metrics, Continuous Inspection

## 1 Why Process Portability Matters

It has never been easier to provision a new and scalable runtime environment for an application than today. In the times of cloud computing, new computing resources can be acquired on demand and set up within seconds. This enables applications to scale up and down intelligently, depending on the workload put onto the system [1].

This flexibility is not for free, but leads to new problems and exacerbates existing ones. One of the problems that regain in importance is the problem of application *portability*. Portability, the ability to move software among different runtime environments without having to rewrite it partly or fully [2, 3], is the prerequisite for benefiting from current trends and a primary enabler for the evolution of process-aware information systems (PAIS). There is no use in being able to provision a new runtime environment, if the target application cannot be adjusted to run in that particular new environment. Application portability is a central characteristic of software quality [4] and is part of various software quality models, e.g., [5–9]. Especially since the arrival of cloud-based applications, work on application portability, e.g., [10, 11], is gaining momentum. This is also demonstrated by recent standardization initiatives, such as the *Topology and Orchestration Specification for Cloud Applications* [12].

Process-aware applications [13] are in a premier position to address these problems and have the potential to better cope with them than traditional applications, as for instance demonstrated by the recent advent of cloud-based process management systems [14]. This results from the fact that the abstraction from the execution platform, i.e., the runtime *engine*, is a fundamental concept in the architecture of process-aware information systems [15]. If a process is developed in a format that is independent of a concrete engine, it should be easily portable to any engine that consumes this format. Major international process standards, such as the *Business Process Model and Notation* (BPMN) [16], the *Web Services Business Process Execution Language* (BPEL) [17], or the *XML Process Definition Language* (XPDL) [18], name the portability of processes as an important goal and provide platform-independent serialization formats that can solve the problem of portability. Theoretically, if a process is implemented in conformance to a standard, it should be executable on and portable to any implementation of the standard. This abstraction can be leveraged in a cloud-based process execution scenario, which is depicted in Fig. 1. Instead of directly deploying a process to a specific engine, a broker agent can automatically select the most suitable one based on metric data and deploy it there. In [19], this example is explained in more detail for the case of BPEL processes and engines. Additionally, the agent can adapt and improve the process before deployment.

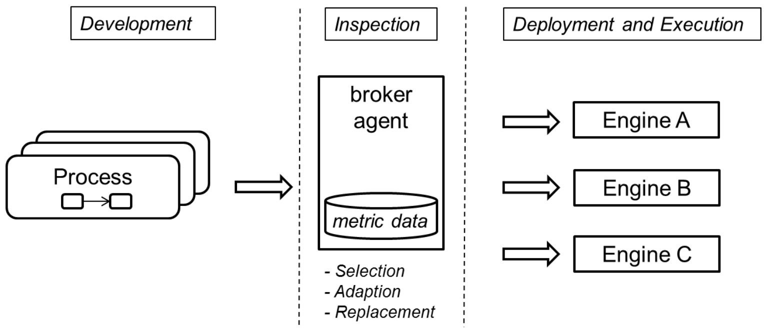

**Fig. 1.** Engine Selection with the Help of a Broker Agent

Despite this extraordinary starting position, process-aware applications today are still limited with respect to their portability. The implementations of international standards rarely support the complete specification, but omit certain parts or differ in the interpretation of the standard [20–22]. As a consequence, even processes that are compliant to a standard might not be portable to any process engine for that standard, if they use features that no engine supports[1]. In other words, it is not possible to depend on a standard alone for achieving process portability. This is an obstacle to PAIS evolution and the application scenario from Fig. 1. In this situation, agile techniques for software quality improvement, such as continuous inspection [24, 25], can be applied as a remedy [24, 26–29]. The prerequisite for the application of such techniques is that portability issues can be detected and quantified. This quantification is the central topic of the current chapter. Work on measuring the portability of applications, and specifically of process-aware ones, is rather scarce. For this reason, we work on a metrics suite for quantifying the portability of and detecting portability issues in process-aware applications, based on recent software quality standards [5]. Such software metrics could be leveraged to support an easier evolution of processes and execution engines. For instance, when a new version of an engine becomes available, metric data can be used to confirm if existing processes can automatically be ported to said engine. As explained in Fig. 1 and outlined in [19], with the help of metric data, an agent could automatically perform the selection of an appropriate engine for a given process. The agent could analyze the process and, based on the constructs it uses, determine on which engines the process can be executed. From this set of possible engines, the most suitable one can be selected based on further quality data or heuristics, such as the ease of its installation. If no suitable engine for a process is found, metrics and inspection methods can be used to identify possibilities for adaptions of the nonportable parts of the process or even replacement candidates for the process as a whole. To sum up, support with software metrics has the potential to reduce the manual effort required for porting processes and to ease the evolution of process-aware information systems.

This chapter is intended as an overview on the topic of portability for process-aware information systems. In Sect. 2.1, we explain in detail why problems of portability are not solved by today's standards as discovered in several case studies. We then go on and describe in Sect. 2.2 and 2.3 how these problems can at least be relaxed through techniques of software measurement and continuous inspection. Thereafter, we give an overview on our current work on software metrics for portability in Sect. 3, along with examples explaining how an intelligent PAIS can make use of metric data. The chapter is based on several publications [3, 20, 21, 30, 31] which we extend and synthesize into a unified context.

## 2 How Software Measurement Can Help

The measurement of software quality is almost as old as the discipline of software engineering itself [32, 33]. The quantification of the quality of a software product can lead to an improvement of its quality [26].

[1] This problem is evident if processes are not implemented manually for a particular engine, but derived automatically, for instance, in a model-driven mapping [23].

In the following subsection, we show why the problem of process portability is not already solved by the international standards and specifications that exist. Thereafter, in Sect. 2.2, we discuss how the usage of agile techniques such as continuous inspection can help to improve this situation, given a mechanism for the detection of portability issues and a quantification of portability is available. This leads to a description of several frameworks that form the basis for this quantification in Sect. 2.3, in particular the ISO/IEC 25010 standard for software quality [5]. The next subsection is partly based on [21].

## 2.1 Standards Are Not Enough

Several standards and notations for building process-aware systems exist today, e.g., [16–18]. It is a popular conception that all that has to be done to achieve portability of process code is to write it in one of these notations. The availability of a standardized serialization format for processes is enough to silence most arguments relating to portability. This is an obvious advantage for the implementers of a standard and vendors of tooling. In the absence of certification authorities, it is easy to claim support for a given standard.

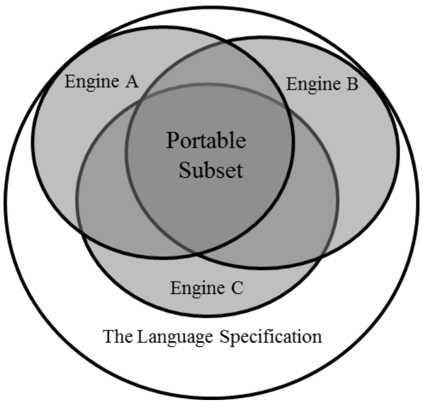

**Fig. 2.** Different Subsets of Supported Language Elements by Different Engines

However, the reality of implementing process-aware applications looks different. Process specifications are complex and might contain ambiguities, as discussed for particular specifications in [22, 34]. Moreover, they often have a large set of language elements. For instance, the BPMN specification lists 63 different types of events in Sect. 10.4 [16]. As a result, the implementers of such a language often just implement a subset of the language or implement some language features in a way that differs from the original language specification. Only the elements of the language that are contained in the overlap of these subsets are truly portable. Other elements are only portable to a limited degree, as depicted in Fig. 2. This implies that a practical porting of a process is often not feasible, despite the fact that multiple implementations claim to support the language the process is implemented in, and portability should therefore theoretically be a given fact.

To shed light on this situation, we performed two studies [20, 21], in which we benchmarked and analyzed a variety of process engines to see how well they support the language they claim to implement. We chose BPEL [17] for these studies. This language has been conceived for service orchestration and has received tremendous interest in academia and industry since the publication of its final version in 2007. Today, this interest is somewhat in decline in favor of other languages, but a variety of runtimes for BPEL have been built and are used in production today. Porting BPEL processes is a real problem practitioners face. This makes an analysis of the runtimes of this language worthwhile.

To see how well process engines support the language, we implemented a conformance benchmarking tool [35], called *betsy*[2]. This tool provides a conformance test suite which covers all language elements of BPEL with more than 130 test cases, in the form of standard-conformant BPEL processes. Additionally, we also provide tests for the functional correctness of implementations of workflow control-flow patterns. Moreover, betsy can be used to automatically manage (download, install, start, and stop) several open source and commercial engines and execute the tests in an isolated fashion [36]. We derived conformance tests from the normative parts of the BPEL specification. The test suite is subdivided into three groups, namely, *basic activities*, *scopes*, and *structured activities*, resembling the structure of the specification listed in sections 10–12 [17]. The various configurations of the BPEL activities of each group form the basis of the test cases, including all BPEL faults. Hence, every test case of the standard conformance test suite asserts the support of a specific BPEL feature. Firstly, every test case consists of a test definition, being the BPEL process definition and its dependencies (WSDL definitions, XML Schemas, etc.). The second part of a test is the test case configuration, being the specification of the input data and assertions on the result. The aim of every test is to check the conformance of a single language feature in isolation. During a full test run, our tool automatically converts the engine independent test specifications to engine specific test cases and creates required deployment descriptors and deployment archives. Next, these archives are deployed to the corresponding engines and the test case configurations are executed. At first, every test case configuration checks successful deployment and, thereafter, performs the different test steps. These test steps send messages and assert the correctness of the responses by means of return values or expected SOAP faults. When all test cases have been executed, HTML and CSV reports are created from the test results. The complete test procedure is quite complex and takes several hours to execute, due to the complexity of the installation of several of the engines. In the end, a comprehensive overview over the support for the BPEL specification is the result. For a more detailed explanation of the testing procedure, we refer the interested reader to [20,21,35].

[2] More information on the tool and a description of its usage is available at the project homepage located at `https://github.com/uniba-dsg/betsy`.

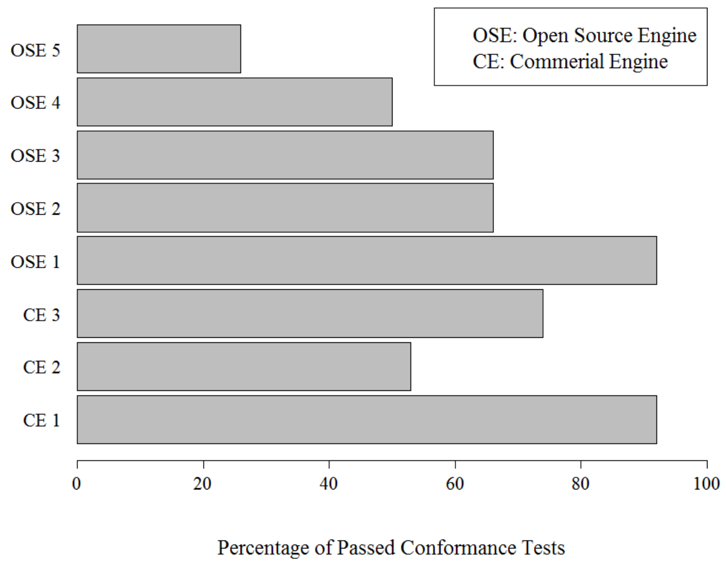

**Fig. 3.** Standard Conformance of Process Engines – The figure displays the percentage of successfully passed tests per engine as discussed in [21].

Fig. 3 provides a rough overview of the state of BPEL support and the most important results from [21]. It shows the percentage of passed tests of our test set for a variety of engines, both of commercial and of open source origin. Two main problems can be read from the plots: Firstly, no engine passes all the tests, hence no engine implements the complete specification. Secondly, the variances in the amount of successful tests is high for the different engines. In more detail, proprietary engines successfully pass between 53% and 92% of the conformance tests. For open source engines, these numbers vary from 26% to 92%. On average, proprietary engines pass 73% of the conformance test suite, whereas the open source engines only achieve 62%. In total, proprietary engines provide a significantly higher degree of support, although the difference balances if we only consider mature open source engines. All in all, the data demonstrate that the porting of processes among these engines will be difficult, due to different degrees of language support.

A similar investigation is currently ongoing for the BPMN language [16] and we extend betsy for benchmarking BPMN engines. Although we cannot yet unveil the detailed results, we can say that the situation is very similar if not even more critical for this language. In any case, the results presented here seem to be representative for the situation we face in practice, regardless of what process language we consider. They demonstrate, that the availability of process standards alone does not guarantee the portability of processes. Implementations will always deviate from a standard for a variety of reasons, such as economic constraints, issues in the specification, or software errors. The best we can do is to adapt to this situation and to try to build portable processes nonetheless. This is necessary to better cope with today's highly dynamic environments and to support the evolution of process-aware information systems. Intelligent support through software measurement and agile techniques such as continuous inspection, which are detailed in the following section, can help in this task.

## 2.2 Quality Improvement With Continuous Inspection

Continuous inspection is a term for the convergence of two quality assurance techniques, *software inspection* [37] and *continuous integration and delivery* [24, 38]. Continuous inspection refers to the constant and automated inspection of a software product for every source code commit to enhance its quality [25]. In this section, we explain how this combination works and why it has the potential for the improvement of process portability.

Software inspections, pioneered by Fagan [37], have a long tradition in software engineering [39, 40]. Ordinarily, these are manual tasks performed as a quality assurance technique next to other techniques such as unit testing. Essentially, an inspection is a review process where a team of reviewers individually scrutinize a software product according to a predefined set of criteria. The reviewers try to verify if the product meets its specifications and has a sufficient level of quality [41]. Afterwards, the reviewers gather in a meeting and produce a list of defects that can be handed to the authors of the software to fix these issues. Obviously, this process requires a lot of communication and is therefore expensive to perform, especially in a repeated fashion. For these reasons, inspection tools have emerged during the last decade. These tools are static code analyzers that automatically highlight potential issues in code [42]. The benefit of their usage is that an inspection can normally be performed within mere seconds and repeatedly. Of course, such a tool might not find all issues or detect false positives, but it offers unprecedented advantages in terms of efficiency. Software inspections need not necessarily be limited to the detection of potential issues, but are also a good occasion to compute quality metrics to see if the software meets predefined quality criteria.

Inspection tools can be used to much benefit, when combined with *continuous integration* (CI) [24]. CI is a term that emerged in the context of agile software development methods. It belongs to the concept of *continuous delivery* [38] which is specified in the first principle of the agile manifesto [43]. CI is a technique applied during software development that refers to the frequent integration of all parts of a program and the validation that they do work together properly. In practice, a continuous integration server is set up and configured to build the program and run the complete set of tests available every time a commit is made to the version control system [24]. This allows getting immediate feedback for the complete team in every stage of development and offers a variety of benefits, as described on pages 17–22 of [38], or 29–32 of [24]. For instance, defects that are newly introduced into the code can be detected immediately and fixed at a point in time where the cost of correction is relatively low. CI has been embraced in practice and a variety of CI servers and tools are available.

With the proper tools, both of these techniques, software inspection and continuous integration, can be combined to continuous inspection [25]. The idea depicted in the feedback cycle in Fig. 4 is to not just run tests for every commit to the version control system, but also to inspect the code automatically with available inspection software. That way, possible defects that are not captured

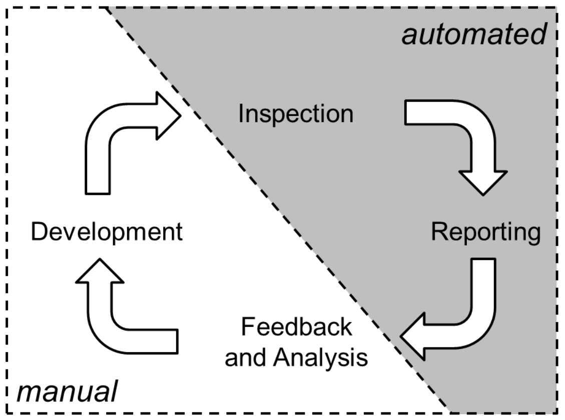

**Fig. 4.** Continuous Inspection Cycle adapted from [25]

in the tests can be discovered and fixed just as quickly. It might even be possible to detect issues in the code that have not yet turned into concrete defects, but are likely to do so in the future. Moreover, this is an opportunity to compute software metrics and compare them to previously configured thresholds [24, 38]. This way, it can be directly perceived if software quality deteriorates and counter measures can be taken before the deterioration turns into software errors. What is more, through this feedback, developers learn which patterns of code tend to reduce quality and which ones tend to improve quality and are encouraged to produce code of higher quality[3], as described on pages 137–140 of [38].

The focus of this chapter is portability. So, applying continuous inspection to this context, this section can be summarized in the following: Given inspection methods and tooling are available for detecting portability issues in code, these methods and tools can be used in today's ubiquitous CI environments through continuous inspection and thus have the potential to lead to higher code quality. In the terms of this chapter, higher code quality refers to more portable code. Here, we present methods and tools for measuring the portability of process-aware information systems. The computation of the metrics we propose is implemented in an open source and freely available inspection and issue detection library, essentially a static analyzer, the *prope* tool[4]. That way, the metrics can be integrated into the continuous inspection cycle. Referring to the

[3] This effect of the influence of the measurement on the persons being measured is known as the *Hawthorne effect*. Although it normally is disruptive for experiments, it can also be used to train developers in the fashion described in the text.

[4] Prope stands for *PROcess-aware information systems Portability mEtrics suite*. For more information on this tool and instructions on how to use it, please visit the project page located at `http://uniba-dsg.github.io/prope/`.

motivating scenario from the introduction depicted in Fig 1, this computation can be leveraged by a broker agent. For every source code commit, a process can be inspected automatically by prope. The inspection results in a list of metric values and issues found in the process, available as CSV and XML reports. An agent can process these reports and use the data to select a proper engine for execution. For instance, engines that do not support one or more activities which the process uses can be immediately excluded from the selection.

## 2.3 Portability and the ISO/IEC 25010 Quality Model

To use a technique such as continuous inspection, it is necessary to be able to capture software quality in the first place. This is where software quality models, e.g., [5–9], come into play. An abundance of quality models has been developed during the last decades[5]. Nevertheless, older quality models, especially by Boehm et al. [6], Gilb [7], or McCall [8], are still very influential. Such models typically define a hierarchy of *quality characteristics* of software, sometimes also called *quality attributes* (cf. Sect. 3.6 of [7]). Each quality characteristic describes a certain major aspect of software quality. Examples are characteristics such as *performance efficiency*, *usability*, or *portability* [5]. It is often useful to divide these high-level characteristics into a number of *subcharacteristics* that focus on more specific aspects[6]. For instance, performance efficiency can be divided into the subcharacteristics of *time behavior*, *resource utilization*, and *capacity* [5]. The main difference between different quality models [5–9] lies in what quality characteristics and subcharacteristics they define and how many layers of characteristics they use.

The quality model used here stems from the ISO/IEC series of quality standards, 9126 [9] and 25010 [5]. The characteristics defined by the model "*are widely accepted both by industrial experts and academic researchers*" [44, p. 68] and often cited (just to mention a few, cf. [4, 26, 45, 46]). This series of standards is very prevalent, since it has been derived from and synthesizes other well-known quality models, e.g., [6–8], and, moreover, has received widespread acceptance in industry due to the standardization process. Software vendors pay a considerable amount of money to obtain an ISO certification.

The ISO/IEC 9126 series [9] is most renowned, but is currently being revised in the context of the ISO/IEC 25010 series [5]. This series is titled "*Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*" and will completely supersede the 9126 series when it is finished. The quality model from [5] is depicted in Fig. 5. It defines eight top-level quality characteristics and one of these is portability, the focus of this chapter. Portability has three subcharacteristics, *adaptability*, *replaceability*, and *installability*. The reasoning behind this structuring can be expressed through a sequence of decisions made when porting an

[5] The amount of quality models has also led researchers to build models of models, such as the systemic quality model presented in [4].

[6] This is called the *attribute hierarchy principle*, defined on page 135 of [7].

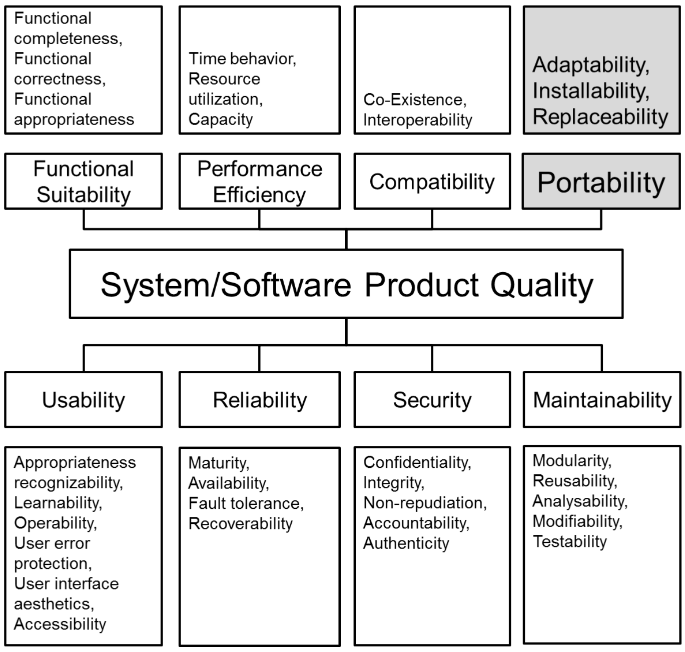
Functional completeness, Functional correctness, Functional appropriateness
Time behavior, Resource utilization, Capacity
Co-Existence, Interoperability
Adaptability, Installability, Replaceability
Functional Suitability
Performance Efficiency
Compatibility
Portability
System/Software Product Quality
Usability
Reliability
Security
Maintainability
Appropriateness recognizability, Learnability, Operability, User error protection, User interface aesthetics, Accessibility
Maturity, Availability, Fault tolerance, Recoverability
Confidentiality, Integrity, Non-repudiation, Accountability, Authenticity
Modularity, Reusability, Analysability, Modifiability, Testability

**Fig. 5.** The ISO/IEC 25010 Quality Model adapted from [5, p.4]

application, as depicted in Fig. 6. If a software product, in our case a process, needs to be ported, the starting point is to check if the process can be *directly ported* in its current form. If this is not the case, there are basically two options:

1. The nonportable parts of the process can be adapted for the new environment. The ease of this depends on the *adaptability* of the process.
2. The process can be *replaced* as a whole by an alternatively available process that runs on the new platform. This depends on whether a suitable alternative is available.

In any case, the new runtime environment for the process has to be *installed.* [5] clarifies that portability and its subcharacteristics have significant influence on the working life of the maintainers and operators of a system. This is especially true for the installability of the system, since the operators of an organization will be the ones who have to perform the installation. As a consequence, an improvement of portability leads to an improvement of the working life of the operators. Enabling the quantification of these characteristics during development provides developers with feedback that allows them to develop software that is better to operate. This integration of *development and operations* is the central goal of the *DevOps* movement [47]. DevOps is a term for another agile practice that aims at improving IT performance and is strongly related to continuous integration and delivery. It is currently receiving widespread attention and is increasingly adopted in practice [48].

Each of the quality characteristics should be quantified to allow for meaningful decisions. The enabling of this quantification is our long-term goal and the content of the remainder of this chapter. At the time of writing, the specification

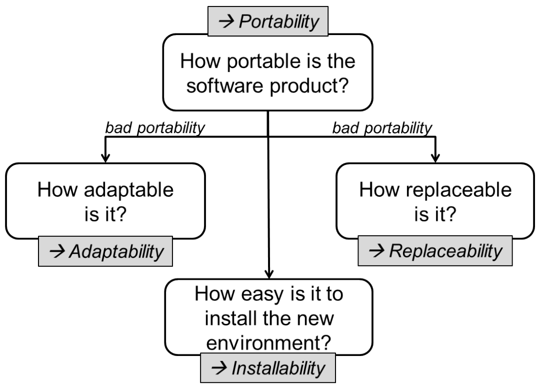

**Fig. 6.** Portability and its Subcharacterisitics

that is to contain concrete metrics [49] is still under development and not yet open to public scrutiny. Moreover, the metrics from preceding versions of the ISO/IEC quality standards [50, 51] are rather general and often based on the observation of human behavior, which is, arguably, not the decisive factor when it comes to the executability of processes on different engines. In previous work, we proposed and validated measurement frameworks for the direct portability of processes [3] and their installability [31]. We are currently working on a similar study for adaptability [30], whereas replaceability still remains open. In the following sections, we give an overview over the proposed frameworks for direct portability and installability with a brief description of the metrics, outline the state of our work for adaptability and provide an outlook on replaceability.

## 3 Metrics for Process Portability

In the following, we discuss how the different characteristics related to portability, *direct portability*, *adaptability*, *replaceability*, and *installability* can be measured for process-aware information systems. We present current metrics, as far as they are available yet. The metric definitions are taken from the respective publications [3, 30, 31].

Our focus lies on the presentation and description of the metrics. Their validation and evaluation is a critical factor, but, due to page constraints, we cannot fully lay out a complete validation and have to refer to the literature [3, 31]. There, we validate the metrics we propose using two theoretical validation frameworks related to *measurement theory* [52] and *construct validity* [53] and complement the validation with an experimental evaluation of process libraries.

### 3.1 Direct Portability

Direct portability is the ability to directly take a piece of software and execute in on another platform without modification. It is a quality characteristic that is typically hard to quantify with reasonable effort [45] and no corresponding metrics for measuring it can be found in the respective ISO/IEC standards [50, 51]. This section is based on [3], where we present an approach for measuring direct portability of processes and evaluate it with BPEL processes.

A practical way for measuring direct portability is by contrasting the complexity of the task of porting a piece of software to the complexity of rewriting it from scratch [45]. To capture this complexity, existing metrics for portability use a lines of code-based calculation. This approach can also be applied to process-based software, but we can improve the accuracy of the measurement by taking domain knowledge of process-aware systems into account. In summary, a portability metric can be based on the following equation:

$$M_{port}(p) = 1 - C_{port}(p)/C_{new}(p) \tag{1}$$

$M_{port}(p)$ is a metric that quantifies the degree of portability for a process $p$. A process can be characterized as a tuple of three sets, $< E, A, S >$, where $E$ is the set of elements of the process, $A$ the set of activities, and $S$ the set of communication activities. Activities are also elements, so $A \subset E$ and also $S \subset A$ applies. $C_{port}(p)$ is the complexity of modifying the process so that it can run on another platform. $C_{new}(p)$ is the complexity of rewriting it completely for a new platform. Equation (1) is based on the assumption that the complexity of a rewrite is always at least as high as the complexity of modification. This implies that the metric value ranges in the interval of zero and one, where zero indicates no portability and one full portability. The difficulty in this equation is how to actually determine the complexity. The different metrics presented here propose different ways of calculating these values.

As described in Sect. 2.1, the direct portability of a process is strongly tailored to the engines that exist for the language the process is written in. Only process elements that are supported by a majority, or all, engines can be considered to be portable. As a consequence, the measurement of process portability should take the engines for said processes into account, and not be based on a theoretical consideration of the problem only. If all available engines support all the existing language elements in the same manner with respect to semantics, then any executable process will be portable to any engine and there are no portability issues. Sect. 2.1 demonstrates that the situation is rather different in practice. Each engine typically supports a specific language subset, as depicted in Fig. 2, causing portability issues. On the one hand, there is a basic subset of the total language that is fully portable. On the other hand, several language elements are only portable in certain configurations or are limited to a subset of engines. The more engines support a language element, the more portable it can be considered.

To take this into account, we calculate a *degree of severity*, referred to as $D_{ta}$, with respect to portability for each language element and its configuration. This degree can be identified by the number of engines that do not support an element. The smaller the amount of engines supporting a language element, the harder it will be to port a process that uses this element. We can precisely determine this amount, by considering the engine benchmarks described in Sect. 2.1.

The benchmarks list for every language element, whether it is supported by a given engine. This enables us to statically check processes for elements that are not supported by all engines, as discovered by the benchmark. The portability metrics we propose describe different aggregations of the support for every language element used in a process to a portability value for the overall process. We consider a high-level view, typical for classical portability metrics, a process-oriented view and a communication-oriented view. In combination, these metrics

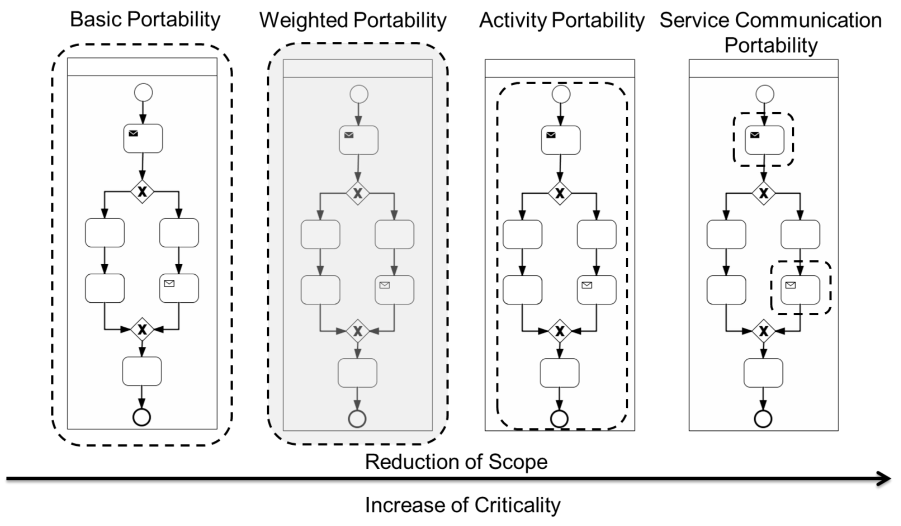

**Fig. 7.** Framework for Measuring Direct Portability – Dashed rectangles mark the scope of a metric.

form a comprehensive framework for quantifying portability, which is depicted in Fig. 7. Although all the metrics focus on direct portability, the scope is reduced for each metric to a more limited, but also more critical part of a process. Simply put, the *basic portability* metric takes all process elements equally into account. The *weighted portability* metric also considers all process elements, but includes engine support data in the computation. *Activity portability* only takes control-flow related aspects, such as activities, gateways, or events, into account. Finally, *service communication* portability only considers activities related to message sending and reception.

**Basic Portability:** A universally applicable way of calculating $C_{port}(p)$ and $C_{new}(p)$, denoted as *basic* portability metric $M_{basic}$, is to consider the lines of code that have to be rewritten for porting the software (as indicated in [6, 7]). If it is to be redeveloped from scratch, all lines will have to be rewritten, so $C_{new}(p)$ amounts to the total lines of code of the program. In our case, lines of code correspond to process elements, so $C_{new}(p)$ refers to the total amount of elements in a process, denoted as $N_{el}$ being the cardinality of set $E$. $C_{port}(p)$ in turn amounts to the elements from $E$ that have to be rewritten when porting it, i.e., the number of elements from $E$ for which problems could be detected. As the number of elements that have to be rewritten for porting cannot be larger than the number of elements that do actually exist, $C_{port}(p) \leq C_{new}(p)$ always applies. In the most extreme case, where all elements are nonportable, $C_{port}(p)$ will be equal to $C_{new}(p)$ and consequently $M_{basic}(p) = 0$, indicating no portability at all. The metric is undefined for an empty program, where $C_{new}(p) = 0$.

**Weighted Portability:** $M_{basic}$ transfers the classical abstract portability metric to the area of process languages. However, it does not make full use of the empirical data at hand. To be precise, it only confronts the amount of fully portable elements of a process to all of them. Using the degree of severity $D_{ta}$, described at the beginning of this section, it is possible to fine-tune this observation, resulting in a more accurate metric value. This is the principle underlying this and the following metrics.

For the *weighted elements* portability metric, the complexity of rewriting a process $C_{new}$ is defined as $C_{new}(p) = N_{el} * N_{engines}$. It is identical to the amount of elements $N_{el}$ (as in the basic portability metric) multiplied with the number of engines under consideration $N_{engines}$. Effectively, every element is treated as if it is unsupported by any engine and has to be rewritten when being ported, resembling the worst case. The complexity of porting $C_{port}$ is defined as follows:

$$C_{port}(p) = \sum_{i=1}^{N_{el}} C_{el}(el_i) \tag{2}$$

The complexity of porting $C_{port}$ of a process $p$ is the sum of the element complexity $C_{el}$ for each element $el_i$ from $E$. The element complexity $C_{el}$ for an element $el_i$ of process $p$ refers to the most severe portability issue that can be detected for the element[7]. It corresponds to the degree of severity as defined above, the number of engines not supporting the element in its current configuration. The more engines that support the feature, the less the complexity of porting it will be. Similarly, the fewer the number of engines, the higher the complexity. Summarizing the above discussion, the weighted elements metric $M_{elem}$ is calculated as follows: $M_{elem}(p) = 1 - \sum_{i=1}^{N_{el}} C_{el}(el_i)/(N_{el} * N_{engines})$.

**Activity Portability:** The most central building block of process languages in general are activities. Activities are typically basic atomic steps of computation. For process complexity measures [54, 55], activities and the transitions among them are the dominant factor. Apart from activities, processes include a variety of other elements such as, for instance, variable definitions. Considering the conceptual importance of activities, it can be expected that the impact of using problematic activities on portability is critical. Having to alter the flow of control for porting a process affects its behavior which is not desirable.

[7] This is a simplified description of element complexity. For an extensive discussion and formal definition of this function, please refer to [3].

An activity-oriented view on portability is provided by the *activity* portability metric $M_{act}$ as a variation of the weighted elements metric. Here, instead of elements, we only consider activities and problematic configurations thereof (i.e., the elements of set $A$) when computing portability. Issues that cannot be linked to a specific activity, as for example process-level *import* statements or variable definitions, are omitted in the consideration of this metric. For $M_{act}$, $C_{new}$ changes to $C_{new}(p) = N_a * N_{engines}$ where $N_a$ denotes the total amount of activities, the cardinality of $A$, in the process definition. $C_{port}$ changes to $C_{port}(p) = \sum_{i=1}^{N_a} C_{el}(a_i)$. This means that only the element complexity $C_{el}$ of the activities in $p$ is considered.

**Service Communication Portability:** Communication and composition relations among services and processes are a decisive factor for service-oriented and process-aware systems and metrics for such systems center on these properties [56]. Communication relationships describe the *observable behavior* of a process; that is, the messages it sends and receives. The distinction between the description of observable and internal behavior is the discriminating factor for different types of process models [57]. Message sending and reception is performed using specific activities. In terms of portability, these activities are most critical. Single elements and perhaps even the control-flow structure of a process may be changed for porting in a way that does not affect the observable behavior. However, this is unlikely if the activities that have to be changed concern communication. In this case, these activities directly affect the observable behavior of the process. Changing them (to enable portability) and consequently changing the observable behavior influences other systems that interact with the process, and this way of change propagation is highly undesirable.

The service communication portability metric $M_{serv}$ allows focusing on the impact of communication related activities on portability. For this metric, the calculation of $C_{new}$ and $C_{port}$ is changed to include only the activities relating to service interaction (i.e., the elements of set $S$), that is: $C_{new}(p) = N_s * N_{engines}$ and $C_{port}(p) = \sum_{i=1}^{N_s} C_{el}(s_i)$. Effectively, this is an extension of $M_{act}$ that focuses solely on activities for service interaction. $N_{serv}$ refers to the total amount of activities for service interaction, the cardinality of $S$. $C_{port}$ is limited to only consider the element cost of these activities.

**In Summary:** The metrics presented in this section and depicted in Fig. 7 try to capture the *direct portability* of a process. For more details and a validation and evaluation of the metrics, please refer to [3]. Each of the metrics corresponds to a reduction in scope and an increase in criticality. In combination, they allow for a comprehensive view of the portability of a process. If evaluated in the continuous inspection cycle, as described in Fig. 4, they can be used to immediately detect a deterioration of the direct portability of a process under development. Fig. 8 considers the scenario of automatic engine selection presented in the introduction. A new activity is inserted into process $X$ and, during inspection, an issue is detected in the configuration of this activity, resulting in a decrease of its

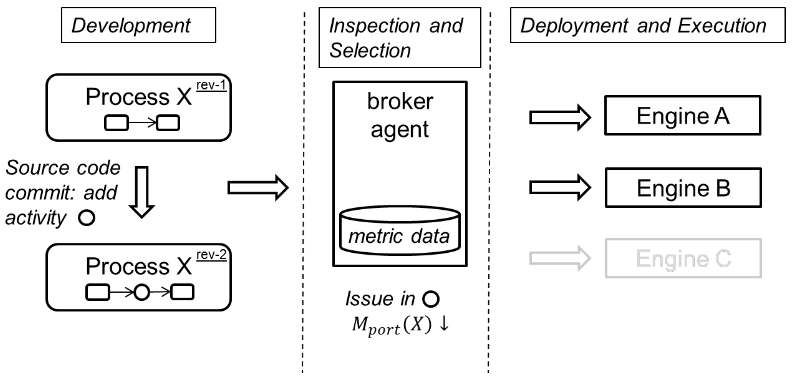

**Fig. 8.** Engine Selection based on Portability Data

portability. The metric values for the process and the issue are reported by the inspection tool, prope, and this information can be utilized by a broker agent. If, for instance, the activity is known to be unsupported by engine $C$, this engine can be excluded from the selection by the broker agent.

## 3.2 Installability

The ISO/IEC SQuaRE model defines installability as the "*degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment*" [5, p. 15]. When it comes to process-aware information systems, two components are of interest with respect to installability: The process itself and its runtime engine. Both of these com-

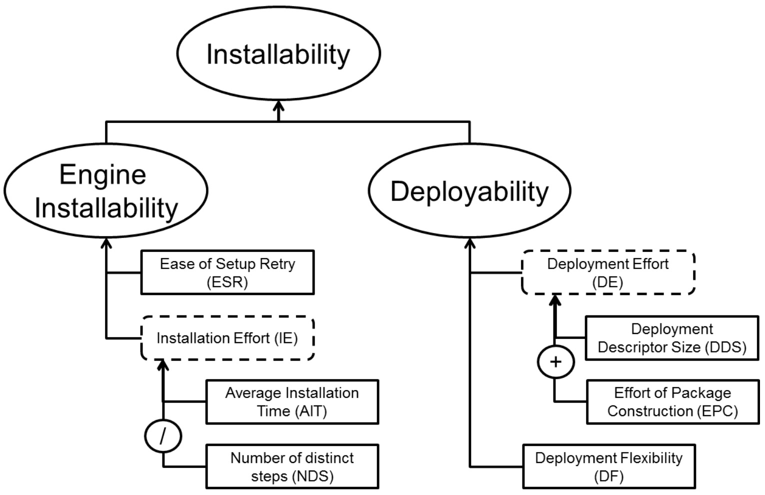
Installability
Engine Installability
Deployability
Ease of Setup Retry (ESR)
Installation Effort (IE)
Average Installation Time (AIT)
/
Number of distinct steps (NDS)
Deployment Effort (DE)
Deployment Descriptor Size (DDS)
+
Effort of Package Construction (EPC)
Deployment Flexibility (DF)

**Fig. 9.** The framework for measuring installability adapted from [31]. Ellipses denote quality characteristics, rectangles denote direct metrics obtained through code analysis and benchmarking, and rounded dashed rectangles depict aggregated metrics that are computed by the combination of direct metrics using the functions displayed in circles.

ponents influence the installability of the complete system and should therefore be taken into account. Fig. 9 outlines the model we use for measuring installability. In [31], on which this section is based, we divide the quality characteristic *installability* into the subcharacteristics *engine installability* and *deployability.*

Each of the subcharacteristics can be measured by a set of direct and aggregated metrics. Direct metrics can be computed directly from source code artifacts or log files, whereas aggregated metrics are formed by the combination of direct metrics. The metrics *ease of setup retry* ($ESR$) and *installation effort* ($IE$) stem from the ISO/IEC standards [50, 51]. We extended installation effort to also consider *average installation time* ($AIT$) and not only the *number of distinct steps* ($NDS$) required for the installation. When it comes to deployability, no corresponding metrics are available in [50, 51], so we develop new ones. These consist of *deployment effort* ($DE$), which considers *deployment descriptor sizes* ($DDS$) and the *effort of package construction* ($EPC$), next to *deployment flexibility* ($DF$). The deployability metrics $DDS$, $EPC$, and $DE$ are *internal* (i.e., they relate to static properties of the software), and the remaining metrics are *external* (i.e., they relate to dynamic properties and can be verified during execution), as specified in Sect. C.3 of [5].

**Ease of setup retry (*ESR*):** This metric, defined in Sect. 8.6.2 of [50], is intended to measure how easy it is to successfully repeat an installation of an engine. It relates the number of successful installations of the same engine $e$ ($N_{succ}$) to the number of attempted installations in total ($N_{total}$). That is $ESR(e) = N_{succ}/N_{total}$. [50] refers to manual installations, but the metric is just as applicable to an automated installation process. If this process is completely deterministic, then those numbers will be identical and $ESR(e)$ equal to one. If it is not free of errors, installations may fail, resulting in a lower $ESR$ value.

**Installation effort (*IE*):** Installation effort provides a notion of the difficulty of the installation process of an engine. Sect. 8.6.2 of [50] suggests to measure it as the amount of automatable installation steps in relation to the total amount of prescribed steps. As we found in the case study described in [31], the installation of process engines can often be automated fully, but the complexity of this automation and, more importantly, the duration of the installation varies a lot. For that reason, we deviate in the measurement of installation effort from [50] and instead measure it through a combination of two direct metrics: The total number of distinct steps ($NDS$) and the average installation time ($AIT$). The first is identical to the number of steps that need to be automated, and thereby partly corresponds to the metric defined in [50]. $NDS$ includes every operation that needs to be performed for the installation, such as the copying of files and creation of directories or changes in the configuration of certain files. This metric can be determined through a *heuristic evaluation* [58], i.e., we can essentially count each step in an installation script. The average installation time can be computed by performing the distinct steps required a suitable amount of times and measuring execution times. $AIT$ and $NDS$ can be aggregated to a notion of installation effort ($IE$) per installation step:

$$IE(e) = \begin{cases} 0 & \text{if } NDS = 0 \\ \frac{AIT(e)}{NDS(e)} & otherwise \end{cases} \tag{3}$$

Note that an installation routine that consists of several simple steps is desirable over a single installation step that takes very long even if the multiple step installation takes longer. The reasoning behind this is that simple and quick installation steps are easier to automate, to repeat in case of a failure, or to adapt to a new environment.

**Deployment Flexibility (DF):** This metric can be used to quantify the availability of alternatives for achieving the deployment of a process. Deployment normally consists of the execution of a single engine operation provided with all artifacts needed for execution. Nevertheless, deployment can take different forms, multiple of which can be supported by an engine. The more options a server supports, the more flexible it is and the easier deployment can be achieved. We capture this in the metric *deployment flexibility* ($DF$), which corresponds to the number of options available. The intention of the metric is to adapt *installation flexibility* from Sect. 8.6.2 of [50] to this context. Typically, three different options are available:
1. Hot deployment, i.e., a copy operation of a deployment archive into a specific directory,
2. the invocation of a deployment script, or web service,
3. a manual user operation using a GUI or web interface.

**Deployment Effort (DE):** Being similar to installation effort, this metric can be used as an overview of the complexity of the preparation of a process for its deployment. Deploying a process normally requires its packaging and the construction of one or more deployment descriptors. The construction of these descriptors may be partly automated or aided by graphical wizards, but in the end it is configuration effort that can take a significant amount of time to get right. The more complex the packaging and the more extensive the descriptors, the harder it is to deploy a process on a specific engine. We capture packaging with the metric *effort of package construction* ($EPC$) and deployment descriptors with the metric *deployment descriptor size* ($DDS$). The effort of package construction can be measured by counting each part of a prescribed folder structure that needs to be built and compression operations that need to be performed to construct the prescribed deployable executable: $EPC(process) = N_{fc} + N_{dc} + N_{co}$. $N_{fc}$ refers to the amount of folder creations, $N_{dc}$ to the amount of descriptors, and $N_{co}$ to the amount of compression operations required. The deployment descriptor size $DDS$ for a process corresponds to the added size of all descriptor files, $\{dd_1, ..., dd_{N_{desc}}\}$, needed: $DDS(process) = \sum_{i=1}^{N_{desc}} size(dd_i)$. For process-aware applications, typically two different types of descriptor files exist in practice: i) plain text files and ii) XML configuration files. As plain text files and XML files differ in the ways in which they represent information, different ways of computing their size are needed. For plain text files, a lines of code metric is appropriate. For the descriptors at hand, every nonempty and noncomment line in such files is a key-value pair with a configuration setting, such as a host or port configuration, needed for deployment. We consider each such line, using a $LOC$ function. For XML files, the notion of lines is not applicable, but instead information is structured in nested elements and attributes. To compute the size of XML files, we consider the *number of elements and attributes* $N_{ea}$, including simple content and excluding namespace definitions, which represent an item of information in the same fashion as key-value pairs in plain text files. All in all, the *size* of a descriptor *desc* is defined as follows:

$$size(desc) = \begin{cases} LOC(desc), & \text{if } plain(desc) \\ N_{ea}, & \text{if } xml(desc) \end{cases} \tag{4}$$

As a result, the deployment effort can be computed in the following fashion: $DE(process) = DDS(process) + EPC(process)$. The idea here is to capture every factor, independent of its nature, that increases the effort of deploying a process.

**In Summary:** The metrics presented above allow for a quantification of the installability of a process-aware information system. Metrics for engine installability can be used to compare different engines and might help to select the best one. Metrics for deployability can be used to compare different processes with each other and are also suitable for continuous inspection. As demonstrated in [31], certain values for deployability are typical for certain engines. Hence, also deployability metrics can be used as a factor when selecting different engines. As

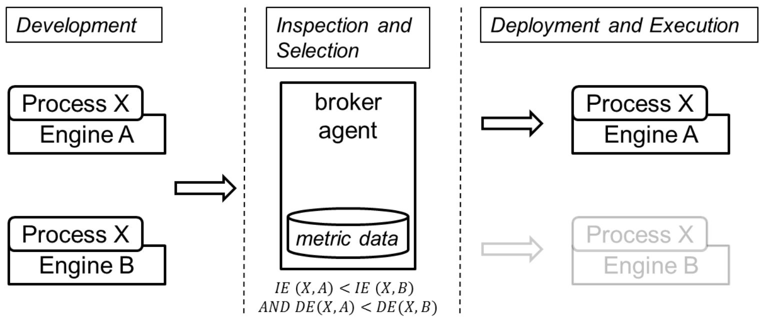

**Fig. 10.** Engine Selection based on Installability Data

before, a more detailed formal definition, as well as a validation and evaluation can be found in [31]. Fig. 10 outlines how this information can be leveraged in the engine selection scenario from the introduction. Prope can be used to compute installability and deployability data of different combinations of a process $X$ and possible engines for $X$. The agent can compare the metric thresholds of the different systems and select the better one, for instance the one with lesser deployment and installability effort.

### 3.3 Adaptability

When it comes to the quality characteristic of adaptability, we did not yet perform extensive evaluations as in [3, 31]. At the time of writing, work on adaptability is underway, but not yet published. Therefore, we give an outline for this quality characteristic and sketch our preliminary approach for computing metrics, presented in [30], but without defining concrete metrics.

The ISO/IEC quality model defines adaptability as the "*degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments*" [5, p. 15]. The definition refers to a manual design-time adaptation. Here, we focus on adaptations to the software environment only. The metrics for adaptability of the existing ISO/IEC quality model [50,51] are based on counting the number of program functions that seem to be adaptable to different contexts. This number is contrasted with the number of functions that are required to be adapted in the current situation, which is typically all program functions that need to be available in the new environment after porting. By relating these two numbers, one can obtain the percentage of program functions that can be adapted. This provides a basic notion of adaptability for the complete program. However, such a measure is coarse and there is no description of how to actually determine if a function is adaptable or not.

Related studies on adaptability metrics are directed at the architectural layer of a software product and not the concrete source code [59, 60]. There, adaptability is first quantified in a binary or weighted fashion for an atomic element of the respective system, such as a component in the software architecture. These element adaptability scores are then subsequently aggregated using different adaptability indices at different layers of abstraction to arrive at a global value of adaptability for the complete software architecture. This way of computing adaptability should also work when looking at code artifacts and not architectural elements of a program. Here, we focus on executable processes and try to reproduce the adaptability computation in the above sense. Thus, our idea is to quantify adaptability at the level of an atomic process element, such as an activity, and to aggregate this to a global degree for the complete process.

For quantifying adaptability at the level of an atomic process element, we count the number of alternative representations for the functionality provided by the element that result in the same runtime behavior. In every process language, there are typically multiple alternatives for each process element that can result in identical process behavior at runtime. The more alternatives exist for a given process element, the easier it is to replace this element with such an alternative, and hence the more adaptable the resulting code actually is.

A simple example for multiple alternative implementations of the same functionality in BPMN [16] is repetitive execution of a task through a *Loop* marker for the task. Any of the following language constructs can be used to define repetitive execution of a task and hence can be used as an alternative to a *Loop* marker:

1. A combination of an *Exclusive Gateway* and *Sequence Flows*
2. Enclosing the task in a *Loop Sub-Process*
3. Enclosing the task in an *Ad-Hoc Sub-Process*
4. Enclosing the task in an *Event Sub-Process*

It is likely that a BPMN engine will only support a subset of these options. For instance, the Activiti engine[8], currently does not support normal *Loop* markers (`standardLoopCharacteristics`). It does support the combination of *Exclusive Gateways* and *Sequence Flows*, as well as *Event Sub-Processes*, but no *Loop* or *Ad-Hoc Sub-Processes*. Given that a process with a task that uses a *Loop* marker needs to be ported to the Activiti engine, the code needs be adapted to one of the versions Activiti supports.

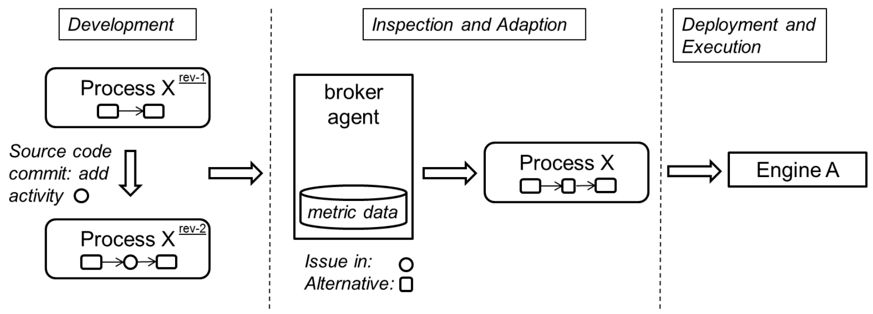

**Fig. 11.** Agent-Aided Process Adaption

Considering the application scenario from the introduction, the design-time adaptation of a process becomes necessary if no suitable engine for executing it can be found. By inspecting the adaptability of the process, in particular with respect to the elements of the process that hinder portability, it is possible to recommend possible adaptions of these elements. Depending on their nature, the adaptions might even be performed transparently and automatically by the broker agent.

In [30], we propose to capture the number of alternative representations of a process element with its *adaptability score*: $AS(e) = \mid \{alt_1^e, \ldots, alt_n^e\} \mid$ This score is equivalent to the cardinality of the set of alternatives $\{alt_1^e, \ldots, alt_n^e\}$ for the element that are available in the language. For the approach to work, such a set of alternatives must be determined for every element of the process language. Based on atomic adaptability scores, a mechanism for aggregating these scores to a global adaptability degree for the complete process is needed. This is necessary to allow for the comparison of different processes in terms of their adaptability. Moreover, the aggregated degree should be normalized with respect to the size of the process, to enable the comparison of processes

[8] For more information on this engine, see the Activiti user guide: `http://www.activiti.org/userguide/index.html`.

of different size. A straightforward way of aggregating adaptability scores is the following:

1. Normalize the score for every element.
2. Similar to [59], compute the mean score of all elements in the process.

This leads to the question of how to normalize scores on an atomic level. We propose to divide the score by a reference value. This reference value can be identified by the maximum adaptability score achieved by any of the elements in the language. That way, the most adaptable language element will have a normalized score of one, whereas other elements will have a value between zero and one. This results in the following equation:

$$AM(p) = \overline{(AS(e_1)/R), \dots, (AS(e_n)/R)} \tag{5}$$

The value of an *adaptability metric AM* of process $p$, which consists of the elements $e_1, \dots, e_n$, is equal to the arithmetic mean of the *adaptability scores AS* for every element $e$ divided by the *reference value R*.

For the choice of the reference value, different schemes are possible. The scheme we use here has several advantages with respect to the computation:

1. The resulting metric value always ranges in the interval of $[0, \dots, 1]$ and thus resembles a percentage value. This scale is easy to understand and interpret, which is critical for the adoption of the metric.
2. The reference value is identical for processes of the same language. Using a reference value that is specific to a concrete process might result in a more meaningful metric value for that process, but it would no longer be directly comparable with different processes. That way, the metric would lose one of its primary purposes.

The division by a reference value is a first proposal for computing an adaptability metric. Alternative ways are possible and we currently test and compare different schemes of computation. We perform an evaluation for BPMN and test the metrics performance for a process library. This way, we hope to find an appropriate way of quantifying adaptability for executable processes. Adaptability metrics computed in this fashion are applicable to continuous inspection.

## 3.4 Replaceability

Also for replaceability, no dedicated metrics have been proposed and tested so far. Hence, we provide a discussion on the nature of the characteristic and present a literature review of existing metrics that could be used for its quantification.

The ISO quality model defines replaceability as the "*degree to which a product can replace another specified software product for the same purpose in the same environment*" [5, p. 15]. This implies that replaceability cannot be evaluated for a single isolated piece of software, but requires a paired combination of two pieces: the currently installed software and a candidate for replacement. In our case, this translates to a currently running process and its replacement candidate. Similar to the case of adaptability, the evaluation of replaceability for a process becomes necessary, when no suitable engine for executing it can be

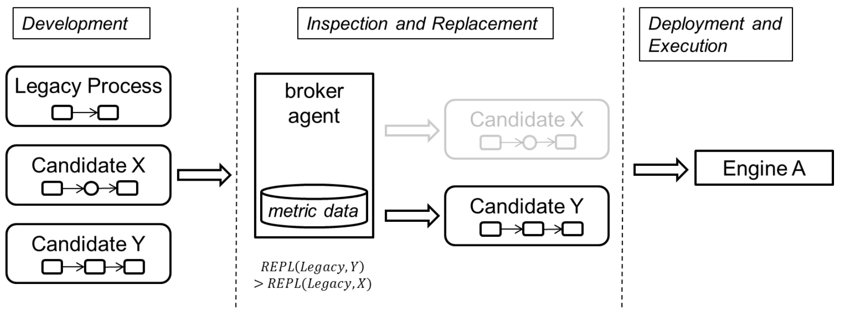

**Fig. 12.** Process Selection based on Replaceability Data

found. Replacing the process as a whole is an alternative to adapting it, but is only possible if a suitable candidate is available, for instance in a process repository. If this is the case, replaceability metrics can be used to determine the best available replacement. This application scenario is depicted in Fig. 12. It might even possible to combine the replacement with adaptability data, to find the most similar process which is the easiest to adapt. This scenario implies that replaceability is typically computed on an ad hoc basis to find an alternative process. In contrast to the other metrics proposed so far, metrics for replaceability are therefore not suitable for continuous inspection. As before, the metrics presented in [50, 51] are solely focused on the observation of user behavior and, hence, not applicable here.

In its core, the question of replaceability is one of *similarity* [61]. Processes are more likely to replace each other if they are highly similar to each other. As a consequence, metrics for process similarity are applicable for evaluating replaceability. Process similarity is important for a wide array of applications apart from replaceability assessment [61], such as process or service discovery [62–65], compliance assurance [66], pattern support assessment [67], and the facilitation of process change [68], just to mention a few. A correspondingly high amount of similarity metrics has been proposed. These metrics measure similarity among processes in terms of *labels*, *structure*, or *behavior* [65]. Labeling approaches quantify similarity by comparing the names of process elements, structural approaches compare the process graph, and behavioral approaches compare execution traces of processes. The abundance of similarity metrics has led researchers to refrain from the definition of new metrics and perform comparative studies on metrics performance instead. A good example of such a study is [69]. In this study, Becker and Laue perform a review of existing metrics, classify them according to the area of application they are directed at and compute metric values for a set of synthetic process models. Based on this computation, they provide suggestions on the quality and appropriateness of the different metrics. For the application of measuring the conformance of executable processes to each other, which is the problem we face here, the authors recommend measures that compute similarity based on the dependencies among the activities of the process graph. They evaluate metrics based on *dependency graphs* [70] and their improvement in *TAR-similarity* [71], *Casual Behavioral Profiles* [72], *Casual Footprints* [64], and the *String Edit Distance of Sets of Traces* [62]. They find casual footprints to be computationally inefficient and criticize the edit distance of sets of traces, TAR-similarity and dependency graphs for considering direct precedence relationships among activities only, and casual behavioral profiles for being unable to handle OR-splits in process models. Here, an additional comparison of these measures with replaceability in mind would be useful.

## 4 Related Work

We tried to discuss approaches related to the work presented here throughout each section. However, three areas should be examined closer:

1. The standards assessment from Sect. 2.1 relates to unit testing and conformance evaluation approaches for process-aware information systems.
2. Software measurement and inspection is, of course, not the only approach for tackling portability issues caused by the dichotomy of specifications and their implementations.
3. A large body of work on software metrics and measurement in general and on metrics for process-aware information systems in particular exists.

In the following three subsections, we discuss each of these areas and provide hints for further reading.

### 4.1 Work on Process Unit Testing and Conformance Validation

The portability metrics we propose build on a conformance testing framework for process-aware information systems, betsy, which relates to other testing approaches in this area. The unit testing of processes, in particular BPEL processes, has received considerable attention [73]. In this area, the *BPELUnit* project [74] is most widely accepted. The main difference between unit testing approaches and the work discussed here is that the former check the correctness of specific process models, whereas we check the correctness of process engines. In other words, the systems under test in our case are different from the systems under test in unit testing approaches.

Conformance checking in the context of process-aware information systems is generally not understood as the testing of the conformance of an engine to the specification it claims to implement. Instead, it refers to the verification of the behavioral properties of a concrete process as specified by an abstract process model. Examples of approaches using this type of conformance checking are [75–78]. Our tool does not check behavioral conformance of concrete process models to abstract specifications. Instead, it checks the implementation conformance of a middleware to a standard specification.

### 4.2 Approaches for Tackling Portability Issues

Studies that address process portability [79–81] also view ambiguities in an informal standard specification as a major problem. [79] try to tackle this problem for BPEL by providing a formal definition of the specification that refines ambiguous aspects. The formalization is accomplished by a formal language called B*lite*. This language can be compiled to executable process code for a specific engine [79]. [80] takes the same approach, by defining a domain specific language that should make programming easier. This approach of pre-compilation can preempt portability problems, by avoiding language elements that are problematic. However, the user of such an approach needs to learn yet another language besides the target one. Here, we do not try to preempt portability issues, but instead to quantify them.

An alternative approach, taken by [81], is to consider the implementation of a standard in practice for improving the standard specification itself. Problems of ambiguity in the specification can be resolved by adopting the interpretation a majority of implementations use in practice. Although we consider the way engines implement the standard in practice here as well, it is not our intent to refine and change the specification, as in [81]. Instead, we determine which aspects of a process definition, although being standard-conformant, cause portability issues and quantify these issues.

## 4.3 Metrics for Selected Quality Characteristics

There is a large body of work on measurement and metrics for process-aware information systems. An overview of the usage of metrics in business process modeling and execution can be found in [82]. General quality metrics for process models build upon classical object-oriented metrics [54], relate to the static complexity of the model during design-time, or the dynamic complexity of the program during run-time [55].

Quality characteristics of processes can often be interpreted, and hence measured, in different ways. In our case, this particularly applies for the quality characteristics of installability, adaptability, and replaceability[9].

Installability, for instance, can also be viewed as the question whether a set of applications can be installed next to each other on the same machine [83]. Component-, or package-based software systems, such as most Linux distributions, are built from package repositories. Software that is installed into the system might require several other packages in particular versions to be installed as well. These package versions can conflict with the versions required by other software, resulting in a failure of the installation. This contrasts the ISO definition of installability, which we build on here. Nevertheless, the contrasting definition of installability is also covered in the ISO quality model, although it is denoted as a different quality characteristic, *co-existence* [5], there.

Deployability of an application can also be considered as the complexity of its deployment into a network of computers. Here, the complexity of deployment relates to the amount of nodes in the network on which an application has to be deployed to function properly [84]. In this definition, the complexity of

[9] As replaceability and its relation to similarity has been discussed as part of the literature review in Sect. 3.4, we omit the repetition of this discussion here.

deploying the application on a single host is not considered. Our point of view on deployability is different here. We do not consider the network-wide deployment of an application, but instead the complexity of deploying it on a single host. This view is more fine-grained, but orthogonal to a network-wide deployment and our framework could be combined with such an approach.

Finally, adaptability is used in a different meaning in several other subject areas. In autonomous systems, adaptability refers to the ability of the system to automatically cope with changing situations, such as an increased load, at run-time [60]. Another definition of adaptability can be found in adapter synthesis. There, adaptability refers to whether an adapter for a pair of services can be created [85].

## 5 Summary and Conclusion

In this chapter, we provided an overview on the topic of process portability. We first demonstrated that a problem of process portability exists for today's process-aware information systems. Thereafter, we described how the problem can be tackled through methods of software measurement and continuous inspection, given metrics for portability are available. Following this, we elaborated on our measurement framework and proposed metrics for quantifying portability, with the subcharacteristics of direct portability, installability, adaptability, and replaceability. Though still under development, this framework provides a holistic quantification of portability and, in combination with measurement tools, has the potential to contribute to process portability in the long term. For instance, a process-aware information system can leverage measurement and inspection tools to intelligently control the deployment of a process on a suitable engine.

Several directions of future work follow. The metrics to be used for quantifying the subcharacteristics of adaptability and replaceability still remain open and suitable ones need to be determined. Further testing and refinement of existing metrics would also be useful. Especially in the areas of direct portability and installability only little work is available and the definition of more metrics and their comparison would be valuable. Moreover, it would be helpful to aggregate metric values for different subcharacteristics to an overall quality indicator for portability. This can be achieved using requirements prioritization methods, such as the analytical hierarchy process [86]. Another area of future work is effort prediction. It would be valuable for practitioners if the effort of porting processes could be estimated upfront. Most effort prediction models estimate effort based on the complexity of the software in terms of lines of code. Our metrics are similar to those approaches, since most of them are based on process elements, a notion similar to lines of code. Therefore, it can be expected that there is a relation between effort and our metric values. Nevertheless, an empirical study would be required to determine this relationship.

## References

1. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A View of Cloud Computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, April 2010.
2. D. Petcu and A. V. Vasilakos, “Portability in clouds: approaches and research opportunities,” *Scalable Computing: Practice and Experience*, vol. 15, no. 3, pp. 251–270, September 2014.
3. J. Lenhard and G. Wirtz, “Measuring the Portability of Service-Oriented Processes,” in *17th IEEE International Enterprise Distributed Object Computing Conference*, Vancouver, Canada, September 2013, pp. 117–126.
4. M. Ortega, M. Pérez, and T. Rojas, “Construction of a Systemic Quality Model for evaluating a Software Product,” *Software Quality Journal*, vol. 11, no. 3, pp. 219–242, 2003.
5. ISO/IEC, *Systems and software engineering – System and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, 2011, 25010:2011.
6. B. Boehm, J. Brown, and M. Lipow, “Quantitive Evaluation of Software Quality,” in *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, USA, October 1976, pp. 592–605.
7. T. Gilb, *Principles of Software Engineering Management.* Addison Wesley, 1988, ISBN-13: 978-0201192469.
8. J. Cavano and J. McCall, “A Framework for the Measurement of Software Quality,” *Software Engineering Notes*, vol. 3, no. 5, pp. 133–140, November 1978.
9. ISO/IEC, *Software engineering – Product quality – Part 1: Quality model*, 2001, 9126-1:2001.
10. D. Petcu, G. Macariu, S. Panica, and C. Crăciun, “Portable Cloud applications – From theory to practice,” *Future Generation Computer Systems, Elsevier*, vol. 29, no. 6, pp. 1417–1430, August 2013.
11. S. Kolb and G. Wirtz, “Towards Application Portability in Platform as a Service,” in *8th International Symposium on Service-Oriented System Engineering*, Oxford, UK, April 2014, pp. 218–229.
12. OASIS, *Topology and Orchestration Specification for Cloud Applications Version 1.0*, November 2013.
13. 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, ISBN: 978-0-471-66306-5.
14. P. Hoenisch, S. Schulte, S. Dustdar, and S. Venugopal, “Self-Adaptive Resource Allocation for Elastic Process Execution,” in *IEEE Sixth International Conference on Cloud Computing*, Santa Clara, CA, USA, June 2013, pp. 220–227.
15. M. Weske, *Business Process Management: Concepts, Languages, Architectures (Second Edition).* Springer-Verlag, Berlin, Heidelberg, 2012, p. Sect. 3.11: Architecture of Process Execution Environments, ISBN: 978-3642286155.
16. ISO/IEC, *ISO/IEC 19510:2013 – Information technology - Object Management Group Business Process Model and Notation*, November 2013, v2.0.2.
17. OASIS, *Web Services Business Process Execution Language*, April 2007, v2.0.
18. WfMC, *Process Definition Interface – XML Process Definition Language*, August 2012, v2.2.
19. S. Harrer, J. Lenhard, G. Wirtz, and T. van Lessen, “Towards Uniform BPEL Engine Management in the Cloud,” in *Proceedings of the CloudCycle14 Workshop.*

Stuttgart, Germany: Gesellschaft für Informatik e.V. (GI), September 2014, pp. 259–270.

20. S. Harrer, J. Lenhard, and G. Wirtz, "BPEL Conformance in Open Source Engines," in *5th IEEE International Conference on Service-Oriented Computing and Applications.* Taipei, Taiwan: IEEE, December 17-19 2012, pp. 237–244.
21. ——, "Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines," in *11th International Conference on Service Oriented Computing.* Berlin, Germany: Springer, Berlin Heidelberg, 2013, pp. 99–113.
22. 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, April 2014, pp. 7–13.
23. J. Lenhard and G. Wirtz, "Detecting Portability Issues in Model-Driven BPEL Mappings," in *25th International Conference on Software Engineering and Knowledge Engineering*, Boston, Massachusetts, USA, June 2013, pp. 18–21.
24. P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley Professional, 2007, ISBN-13: 978-0321336385.
25. P. Merson, A. Aguiar, E. Guerra, and J. Yoder, "Continuous Inspection: A Pattern for Keeping your Code Healthy and Aligned to the Architecture," in *3rd Asian Conference on Pattern Languages of Programs*, Tokyo, Japan, March 2014.
26. C. Jones, *Software Engineering Best Practices – Lessons from Successful Projects in the Top Companies.* McGraw-Hill, 2010, ch. 9. Software Quality: The Key to Successful Software Engineering, ISBN: 978-0-07-162162-5.
27. T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and Software Technology*, vol. 50, no. 9–10s, pp. 833–859, 2008.
28. D. Karlström and P. Runeson, "Combining Agile Methods with Stage-Gate Project Management," *IEEE Software*, vol. 22, no. 3, pp. 43–49, 2005.
29. M. Huo, J. M. Verner, L. Zhu, and M. A. Babar, "Software Quality and Agile Methods," in *28th International Computer Software and Applications Conference*, Hong Kong, China, December 2004, pp. 520–525.
30. J. Lenhard, "Towards Quantifying the Adaptability of Executable BPMN Processes," in *Proceedings of the 6th Central-European Workshop on Services and their Composition*, Potsdam, Germany, February 2014, pp. 34 –41.
31. J. Lenhard, S. Harrer, and G. Wirtz, "Measuring the Installability of Service Orchestrations Using the SQuaRE Method," in *6th IEEE International Conference on Service-Oriented Computing and Applications.* Kauai, Hawaii, USA: IEEE, December 16-18 2013, pp. 118–125.
32. M. H. Halstead, *Elements of Software Science.* Prentice Hall, 1977, ISBN-13: 978-0444002051.
33. T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, December 1976.
34. A. Lapadula, R. Pugliese, and F. Tiezzi, "A Formal Account of WS-BPEL," in *Proceedings of the 10th Internation Conference on Coordination Models and Languages*, Oslo, Norway, June 4-6 2008, pp. 199–215.
35. S. Harrer and J. Lenhard, "Betsy – A BPEL Engine Test System," University of Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, no. 90, July 2012, technical report.
36. S. Harrer, C. Röck, and G. Wirtz, "Automated and Isolated Tests for Complex Middleware Products: The Case of BPEL Engines," in *IEEE Seventh International*

*Conference on Software Testing, Verification and Validation Workshops*, Cleveland, Ohio, USA, April 2014, pp. 390–398.

37. M. Fagan, “Design and Code Inspections to Reduce Errors in Program Development,” *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.
38. J. Humble and D. Farley, *Continuous Delivery.* Addison-Wesley, 2010, ISBN: 0321601912.
39. A. Aurum, H. Petersson, and C. Wohlin, “State-of-the-Art: Software Inspections after 25 Years,” *Software: Testing, Verification and Reliability*, vol. 12, no. 3, pp. 133–154, 2002.
40. M. Ciolkowski, O. Laitenberger, H. D. Rombach, F. Shull, and D. E. Perry, “Software Inspections, Reviews & Walkthroughs,” in *Proceedings of the 22rd International Conference on Software Engineering*, Orlando, Florida, USA, May 2002, pp. 641–642.
41. IEEE, *IEEE Std 1028-2008, IEEE Standard for a Software Reviews and Audits*, 2008, revision of IEEE Std 1028-1997, Sect. 6. Inspections.
42. J. Foster, M. Hicks, and W. Pugh, “Improving Software Quality With Static Analysis,” in *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, San Diego, California, June 2007, pp. 83–84.
43. K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mallor, K. Shwaber, and J. Sutherland, *The Agile Manifesto*, 2001.
44. R. Ferenc, P. Hegedűs, and T. Gyimóthy, “Software Product Quality Models,” in *Evolving Software Systems*, T. Mens, A. Serebrenik, and A. Cleve, Eds. Berlin Heidelberg: Springer, 2014, ISBN: 978-3-642-45397-7.
45. M. Glinz, “A Risk-Based, Value-Oriented Approach to Quality Requirements,” *IEEE Computer*, vol. 25, no. 8, pp. 34–41, March/April 2008.
46. B. Behkamal, M. Kahani, and M. K. Akbari, “Customizing ISO 9126 quality model for evaluation of B2B applications,” *Information and Software Technology*, vol. 51, no. 3, pp. 599–609, 2009.
47. M. Loukides, *What is DevOps?* O’Reilly Media, 2012, ISBN: 1-4493-3910-7.
48. Puppet Labs, IT Revolution Press, ThoughtWorks, *2014 State of DevOps Report*, June 2014.
49. ISO/IEC, *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality*, 2013, 25023.
50. ——, *Software engineering – Product quality – Part 3: Internal metrics*, 2003, 9126-3:2003.
51. ——, *Software engineering – Product quality – Part 2: External metrics*, 2003, 9126-2:2003.
52. L. Briand, S. Morasca, and V. Basily, “Property-based software engineering measurement,” *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.
53. C. Kaner and W. Bond, “Software Engineering Metrics: What Do They Measure and How Do We Know?” in *10th International Software Metrics Symposium*, Chicago, USA, September 2004, pp. 1–12.
54. I. Vanderfeesten, J. Cardoso, J. Mendling, H. Reijers, and W. van der Aalst, *Quality Metrics for Business Process Models.* Future Strategies, May 2007.
55. J. Cardoso, “Business Process Quality Metrics: Log-Based Complexity of Workflow Patterns,” in *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS.* Springer, 2007, pp. 427–434.

56. H. Hofmeister and G. Wirtz, “Supporting Service-Oriented Design with Metrics,” in *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, Munich, Germany, September 2008, pp. 191–200.
57. C. Peltz, “Web Services Orchestration and Choreography,” *IEEE Computer*, vol. 36, no. 10, pp. 46–52, October 2003.
58. J. Nielsen, *Usability Inspection Methods*, 1994, Wiley, New York, ISBN: 978-0471018773.
59. N. Subramanian and L. Chung, “Metrics for Software Adaptability,” in *Software Quality Management Conference*, Loughborough, UK, April 2001.
60. D. Perez-Palacin, R. Mirandola, and J. Merseguer, “On the Relationships between QoS and Software Adaptability at the Architectural Level,” *Journal of Systems and Software*, vol. 87, no. 1, pp. 1–17, 2014.
61. Z. Zhou, W. Gaaloul, F. Gao, L. Shu, and S. Tata, “Assessing the Replaceability of Service Protocols in Mediated Service Interactions,” *Future Generation Computer Systems*, vol. 29, no. 1, pp. 287–299, 2013.
62. A. Wombacher and C. Li, “Alternative Approaches for Workflow Similarity,” in *Seventh International Conference on Services Computing*, Miami, Florida, USA, July 2010, pp. 337–345.
63. R. M. Dijkman, M. Dumas, and L. García-Bañuelos, “Graph Matching Algorithms for Business Process Model Similarity Search,” in *7th International Conference on Business Process Management*, Ulm, Germany, September 2009, pp. 48–63.
64. R. M. Dijkman, M. Dumas, B. F. van Dongen, R. Käärik, and J. Mendling, “Similarity of Business Process Models: Metrics and Evaluation,” *Information Systems*, vol. 36(2), pp. 498–516, 2011.
65. M. Kunze, M. Weidlich, and M. Weske, “Behavioral Similarity – A Proper Metric,” in *9th International Conference on Business Process Management*, Clermont-Ferrand, France, August, September 2011, pp. 166–181.
66. P. Fettke, P. Loos, and J. Zwicker, “Business Process Reference Models: Survey and Classification,” in *Business Process Management Workshops*, Nancy, France, September 2005, pp. 469–483.
67. J. Lenhard, A. Schönberger, and G. Wirtz, “Edit Distance-Based Pattern Support Assessment of Orchestration Languages,” in *19th International Conference on Cooperative Inforamtion Systems*, Hersonissos, Crete, Greece, October 2011, pp. 137–154.
68. C. Li, M. Reichert, and A. Wombacher, “On Measuring Process Model Similarity Based on High-Level Change Operations,” in *27th International Conference on Conceptual Modeling*, Barceclona, Spain, October 2008, pp. 248–264.
69. M. Becker and R. Laue, “A Comparative Survey of Business Process Similarity Measures,” *Computers in Industry*, vol. 63, no. 2, pp. 148–167, 2012.
70. J. Bae, J. C. L. Liu, and W. B. Rouse, “Process Mining, Discovery, and Integration using Distance Measures,” in *International Conference on Web Services*, Chicago, USA, September 2006, pp. 479–488.
71. H. Zha, J. Wang, L. Wen, C. Wang, and J. Sun, “A workflow net similarity measure based on transition adjancency relations,” *Computers in Industry*, vol. 61, no. 5, pp. 463–471, 2010.
72. M. Weidlich, J. Mendling, and M. Weske, “Efficient Consistency Measurement Based on Behavioral Profiles of Process Models,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 410–429, 2011.
73. Z. Zakaria, R. Atan, A. Ghani, and N. Sani, “Unit Testing Approaches for BPEL: A Systematic Review,” in *16th Asia-Pacific Software Engineering Conference*, Penang, Malaysia, December 2009, pp. 316–322.

74. D. Lübke, “Unit Testing BPEL Compositions,” in *Test and Analysis of Service-oriented Systems.* Springer, 2007, pp. 149–171, ISBN 978-3540729112.
75. 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*, Brisbane, Australia, September 2007, pp. 139–153.
76. 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. 29–59, May 2008.
77. A. Both and W. Zimmermann, “Automatic Protocol Conformance Checking of Recursive and Parallel BPEL Systems,” in *Sixth IEEE European Conference on Web Services*, Dublin, Ireland, November 2008, pp. 81–91.
78. 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 & Industrial Conference – Practice and Research Techniques*, Windsor, United Kingdom, August 2006, pp. 127–130.
79. L. Cesari, A. Lapadula, R. Pugliese, and F. Tiezzi, “A tool for rapid development of WS-BPEL applications,” in *25th ACM Symposium on Applied Computing, Sierre, Switzerland*, March 2010, pp. 2438–2442.
80. B. Simon, B. Goldschmidt, and K. Kondorosi, “A human readable platform independent domain specific language for bpel,” in *Second International Conference on Networked Digital Technologies*, Prague, Czech Republic, July 2010, pp. 537–544.
81. T. Hallwyl, F. Henglein, and T. Hildebrandt, “A standard-driven implementation of WS-BPEL 2.0,” in *25th ACM Symposium on Applied Computing, Sierre, Switzerland*, March 2010, pp. 2472–2476.
82. L. S. González, F. G. Rubio, F. R. González, and M. P. Velthuis, “Measurement in business processes: a systematic review,” *Business Process Management Journal*, vol. 16, no. 91, pp. 114–134, January 2010.
83. R. D. Cosmo and J. Vouillon, “On Software Component Co-Installability,” in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Szeged, Hungary, September 2011, pp. 256–266.
84. IETF, *Metrics for the Evaluation of Congestion Control Mechanisms*, March 2008, IETF Network Working Group.
85. Z. Zhou, S. Bhiri, H. Zhuge, and M. Hauswirth, “Assessing Service Protocol Adaptability Based on Protocol Reduction and Graph Search,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 9, pp. 880–904, June 2011.
86. J. Karlsson, C. Wohlin, and B. Regnell, “An Evaluation of Methods for Prioritizing Software Requirements,” *Information and Software Technology*, vol. 39, no. 14–15, pp. 939–947, 1998.