Measuring the Portability of Executable Service-Oriented Processes

Jörg Lenhard and Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
{*joerg.lenhard,guido.wirtz*}*@uni-bamberg.de*

***Abstract*—A key promise of process languages based on open standards, such as the *Web Services Business Process Execution Language*, is the avoidance of vendor lock-in through the portability of process definitions among runtime environments. Despite the fact that today, various runtimes claim to support this language, every runtime implements a different subset, thus hampering portability and locking in their users. In this paper, we intend to improve this situation by enabling the measurement of the degree of portability of process definitions. This helps developers to assess their process definitions and to decide if it is feasible to invest in the effort of porting a process definition to another runtime. We define several software quality metrics that quantify the degree of portability a process definition provides from different viewpoints. We validate these metrics theoretically with two validation frameworks and empirically with a large set of process definitions coming from several process libraries.**

***Keywords*-SOA, BPEL, portability, metrics**

I. Introduction

Software portability is a central characteristic of software quality [1] and is part of various software quality models [2], [3], [4]. Portability is the ability to move software among different runtime platforms without having to rewrite it partly or fully. Service-oriented systems address the communication and interoperation of heterogeneous runtimes [5]. Because of this diversity in runtime platforms, portability is of major importance and is the prerequisite for building truly agile and flexible systems that do not lock in their users. Especially since the arrival of cloud-based services, work on services portability is gaining momentum, as demonstrated by standardization initiatives for portability, such as the *Topology and Orchestration Specification for Cloud Applications*[1] [6] by OASIS.

Portable services can be built by implementing them in a platform-independent language. Process languages address the aspect of portability and recent languages [7], [8], [9] make heavy use of services technology. The *Web Services Business Process Execution Language* (BPEL) [7], which can be used to build stateful services by orchestrating other services using a control- and data-flow definition between the different invocations, provides a platform-independent format and has attracted a huge amount of interest in service-oriented computing. In previous work [10], we presented a tool for benchmarking BPEL engines and could show that current engines are limited in terms of their support for the language specification and the portability of process definitions among them cannot be taken for granted. As a consequence, even services that are designed to be portable by being implemented in BPEL, tend not to be so. In this paper, we mitigate this problem by making the degree of portability provided by a process definition quantifiable. Thus, we support the development of portable process definitions. We define, formalize and validate a set of metrics for assessing process definitions, in contrast to engines as in [10]. Benchmarking results from the tool are one of several factors fed into the metrics computation (cf. section III). Finally, the data set used in this paper is a considerable extension of the data underlying [10], by roughly 30%. Although we build on BPEL, the approach we take for measurement and the metrics we propose are independent of it and applicable to service-oriented processes in general. Specifically, we are trying to answer the question:

How to measure the degree of portability of an executable service-oriented process?

Even though portability is recognized as a quality property of software for a long time [3], [4], it is hard to quantify with justifiable effort [11]. In general, it is measured by contrasting the effort required for porting a piece of software to the effort of rewriting it from scratch. Determining this effort empirically is difficult and an automated calculation is desirable. For this reason, existing and universally applicable metrics use a lines of code-based calculation. However, they are rather coarse and their meaningfulness is limited. Here, we take into account domain knowledge of the languages and environments the programs are written in and run on to provide a more accurate measurement of portability. We map existing metrics for program portability to service- and process-oriented programs, in particular BPEL process definitions, and define metrics that consider the typical characteristics of these programs. We combine the metrics with empirical data on language support in current runtimes as a crucial ingredient of the metrics calculation. The feasibility of their computation is demonstrated in a proof-of-concept prototype that calculates the metrics for existing

[1]See http://www.tosca-open.org/ for more information. Revision 1.0 [6] suggests to use BPEL as one alternative language for defining plans.

programs. Finally, we assess the validity of the metrics from a theoretical point of view using two validation frameworks [12], [13] and complement the validation with an evaluation of four process libraries.

In the next section, we outline related work. Thereafter, we present our methodology and formal definitions of the metrics. Next, they are validated theoretically and empirically. Finally, we present areas of future work.

II. Related Work

Process definition languages and BPEL [7] have attracted a lot of interest in service-oriented systems during the last decade. Related work separates in three areas: i) work on process languages, executable or not, for building and porting process-oriented programs, ii) approaches that address portability issues of BPEL programs specifically, and iii) metrics for measuring properties of service-oriented systems and portability of software in general.

A. Related Process Languages

Considerable effort has been put into XML-based process languages like BPEL. Notable competitors are the *XML Process Definition Language* 2.2 (XPDL) [9] and the *Business Process Model and Notation* 2.0 (BPMN) [8]. Enabling the portability of processes among editors and runtimes is the very reason for these languages to provide an XML serialization format. Each of them has specific areas of focus. We must emphasize that our focus here is executable process definitions; that is actually executable programs.

The focus of XPDL is the storage and interchange of process models. This means, it is specifically tailored to porting process models between tools of different vendors. However, its main purposes are documentation, monitoring, and simulation [9, p. 10] and not primarily execution. For that reason, and a lack of runtimes that use XPDL as execution language, we address BPEL instead of XPDL. Basically, the same applies to BPMN, as its main focus is visualization and communication. BPMN does provide an XML serialization format and addresses execution semantics of process definitions in its current revision 2.0, but actual BPMN runtimes are not yet widely available[2]. BPEL has been in place for several years longer which means that a larger set of runtimes is available and, as indicated by interoperability issues in recent BPMN runtimes [14], these have reached a higher state of maturity. This maturity makes the observation of portability problems, which despite the time still exist, more valuable.

Although we operationalize the approach for measuring portability in this paper with BPEL, it is language-independent. So, it can be tailored to BPMN or XPDL, given that these languages become of major importance for building directly executable service-oriented processes.

B. Approaches Addressing Portability Issues in BPEL

The BPEL specification is an informal standard specification and as a consequence, it is not completely free of ambiguities. According to [15], this is a major reason for why portability issues do exist. [15] try to tackle this problem by providing a formal definition for BPEL that refines ambiguous aspects. The formalization is accomplished by a formal language called B*lite*. This language can be compiled to executable BPEL for a specific engine [15]. This approach of pre-compilation can preempt portability problems. However, the user of such an approach needs to learn another language besides BPEL. Here, we do not try to preempt portability issues, but instead to quantify them. We do not define a new language, but propose metrics for calculating the effort required to port code.

An alternative approach, taken by [16], is to consider the implementation of the 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 runtimes use in practice. Although we consider the way runtimes implement the standard in practice here as well, it is not our intent to refine and change the specification, as in [16]. Instead, we determine which aspects of a process definition, although being standard-conformant, cause portability issues and quantify these issues.

C. Services and Processes Metrics

Service-oriented systems have been addressed by classical object-oriented metrics and evaluations. Examples are cohesion and coupling metrics coming from object-oriented design [17], [18]. Also performance metrics, such as throughput and response times [19], [20], have attracted a lot of interest. To the best of our knowledge, portability has not yet been measured for service-oriented systems.

An overview of the usage of metrics in business process modeling and execution can be found in [21]. Quality metrics for process models also build upon classical object-oriented metrics [22], relate to the static complexity of the model during build-time, or the dynamic complexity of the program during run-time [23]. Complexity metrics specifically tailored to BPEL also exist [24]. Measuring portability, however, has been neglected so far.

Portability is notoriously hard to measure in a quantitative fashion [11]. Original definitions of portability metrics [3], [4] are still valid today and indicated in international quality standards of the ISO/IEC series [25][3]. This series also defines several subcharacteristics of portability, being *installability*, *replaceability*, and *adaptability*. Here, we limit

[2]An example of a BPMN runtime is Activiti. Most vendors, however, do not use BPMN as an execution language. Instead, they use it to visualize a process and map the visualization to another language for execution. Often, BPEL or a proprietary dialect of it are used as the execution platform. This approach is for example applied in ActiveVOS or Oracle SOA Suite.

[3]ISO/IEC CD 25023 is intended to revise the preceding ISO/IEC standards that define quality metrics, ISO/IEC TR 9126-2:2003 and ISO/IEC TR 9126-3:2003, and at the time of writing is still under development.

the scope to the *direct portability* of program code among runtime environments, but the consideration of replaceability or adaptability of nondirectly portable code is complementary to our approach. Direct portability can be measured as the relation between the effort to port the software to a new platform and the effort to rewrite it newly from scratch. Although this approach of quantifying portability is not difficult to understand, the computation of the effort values is. Moreover, the quantification is language- and system-dependent, and requires a high degree of in-depth knowledge. For these reasons, the computation of existing portability metrics is often not feasible or, if performed in a high-level language-independent fashion, not very meaningful. Our contribution here is to provide several metrics that are specifically tailored to BPEL process definitions. The metrics are constructed based on language runtime benchmarks which increases their meaningfulness and relevance to practice.

III. Measuring Portability

The portability of a program is strongly tailored to the runtimes of that program. Only program elements that are supported by a majority, or all, runtimes can be considered to be portable. As a consequence, the measurement of the portability of executable process definitions should take the runtimes for said process definitions into account, and not base on a theoretical consideration of the problem only. If

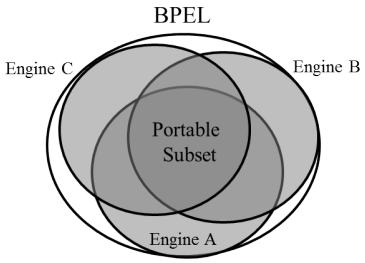

Figure 1. Schema of Language Support

all runtimes available support all of the language elements available in the same manner with respect to semantics, then any compilable program will be portable to any runtime and there are no portability issues. Language runtimes for Java come close to this property, but for BPEL and service-oriented process languages in general, the situation is rather different, as demonstrated in recent benchmarks [10]. There, each runtime typically supports a specific language subset, as outlined in Fig. 1, 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 runtimes. The more runtimes support a language element, the more portable it can be considered.

The first step towards measuring the portability of a process is to calculate the degree of portability for each language element and its configuration. This degree can be identified by the number of runtimes that support an element (i.e., to determine the subset the element occupies in Fig. 1). The smaller the amount of runtimes supporting a language element, the harder it will be to port code that uses this element. To calculate this number for all language elements of BPEL, we performed a benchmark of the BPEL support of seven BPEL engines using the tool *betsy*[4] [26]. The benchmark comprises the engines ActiveBPEL, bpel-g, Apache ODE, OpenESB Sun BPEL Service Engine, Orchestra, EasyBPEL, and a proprietary engine from a major middleware vendor, whose identity we cannot disclose for licensing reasons. It produces a data set which lists for every language element of BPEL, whether it is supported by a given engine. It could also be beneficial to consider combinations of language elements, because certain combinations might result in added portability issues. So far however, we could not find such combinations and, therefore, consider only elements in isolation here. The benchmark is fully automated and reproducible, so this data can be replicated. This enables us to statically check BPEL process definitions for elements that are not supported by all engines, as discovered by the benchmark. The portability metrics we propose in the following section describe different aggregations of the support for every language element used in a process definition to a portability value for the overall process definition.

We have implemented a static checker, the *bpp* tool, that uses the results from the benchmark to automatically check BPEL code and calculate portability metrics[5]. The tool works similar to the testing tools of the Web Services Interoperability Organization. It uses a scheme of test assertions to statically check process definitions for problematic elements. Each test assertion defines a normative requirement that should be respected to achieve portability. There is one test assertion for every problematic language element or configuration thereof. Additionally, each assertion is associated with a *degree* (D_{ta}), which is a quantitative representation of its severity in terms of portability. The degree represents the number of engines that do *not* support the feature (i.e., language element or specific configuration thereof) an assertion is checking. That means, the degree identifies the position of the language element in Fig. 1. A high degree value means that the usage of the language element is an obstacle to portability. With this calculation, all engines are considered of equal practical importance, which is a questionable assumption. That way, the impact of experimental engines is larger than justified from their practical usage. As we lack independent data on engine

[4]Betsy is a conformance testing tool for BPEL. For more information, see https://github.com/uniba-dsg/betsy. The tool has also been used in [10]. The data underlying this paper is an extension of [10] by two engines.

[5]See the project page https://github.com/uniba-dsg/bpp for more information and a description on how to use the tool.

usage, however, we cannot construct an objective weighting here. Therefore, we consider the equal weighting of all engines a reasonable compromise.

IV. Portability Metrics

In the following sections, we present several metrics that measure the portability of BPEL process definitions from different viewpoints. These are a high-level view, typical for classical portability metrics, a process-oriented view and a service-oriented view. In combination, these metrics form a comprehensive framework for quantifying portability.

The scheme of calculating metrics explained in the previous section uses empirical data as a crucial ingredient for the weighting of the metrics. If this data, describing the language support in engines, changes, also the metrics values change. We claim that this is valuable, because it takes into account the evolution of runtimes which are the decisive factor for program portability, and produces more meaningful results than purely theoretically founded metrics.

A. Basic Portability Metric

As discussed in related work (cf. section II-C), portability metrics quantify the relation between the *cost* or *effort* of porting software and rewriting it from scratch [11]. As such, a portability metric for service-oriented processes can be based on the following equation:

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

$M_{port}(p)$ is a metric that quantifies the degree of portability for a process definition p. A process definition can be characterized as a tuple of three sets, $< E, A, S >$, where E is the set of elements of the process definition, 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. This distinction is necessary for the following metrics. $C_{port}(p)$ is the cost of modifying the process definition so that it can run on another platform. $C_{new}(p)$ is the cost of rewriting it completely for a new platform. Equation (1) is based on the assumption that the cost of a rewrite is always at least as high as the cost 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 cost. The different metrics presented here propose different ways of calculating these values. Realistically, the cost of porting or rewriting software is also dependent on the skill of the people carrying out this task. A precise effort prediction for a certain team, would require the extension of the metric with a multiplying factor that characterizes the skill of the team. Here, we abstract from this human aspect and omit such a multiplier.

A universally applicable way of calculating $C_{port}(p)$ and $C_{new}(p)$, which we denote as the *basic* portability metric M_b, is to take into account the lines of code that have to be rewritten for porting the software (as indicated in [3], [4]). 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. $C_{port}(p)$ in turn amounts to the lines of code that have to be rewritten when porting it. As the number of lines that have to be rewritten for porting cannot be larger than the number of lines that do actually exist, $C_{port}(p) \leq C_{new}(p)$ always applies. In the most extreme case, where all lines are nonportable, $C_{port}(p)$ will be equal to $C_{new}(p)$ and consequently $M_b(p) = 0$, indicating no portability at all. In the other extreme, no line will have to be rewritten and $M_b(p) = 1$. The metric is undefined for an empty program, where $C_{new}(p) = 0$.

To automatically calculate $C_{port}(p)$ and $C_{new}(p)$ in this setting, one more assumption has to be made. The language under consideration is an XML dialect and as such abstracts from the notion of lines of code. Instead, XML elements are the crucial unit, identifying a single statement or instruction. For that reason, we base the calculations on XML elements instead of lines of code. For M_b, $C_{new}(p)$ refers to the total amount of elements in a process definition, denoted as N_{el} being the cardinality of set E, and $C_{port}(p)$ to the number of elements from E for which problems could be detected.

B. Weighted Elements Portability Metric

M_b transfers the classical abstract portability metric [11] to the area of XML-languages. However, it is not tailored to service-oriented or process-oriented programs and 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 definition to all of them. Using the degree D_{ta} (cf. section III), it is possible to relativize this observation, resulting in a more accurate metric value. This is the principle underlying this and the following metrics.

The *weighted elements* portability metric M_e takes the degree D_{ta} of elements into account. Here, the cost of rewriting a process definition C_{new} is defined as follows:

$$C_{new}(p) = N_{el} * N_{engines} \tag{2}$$

This cost 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 cost of porting C_{port} is defined as follows:

$$C_{port}(p) = \sum_{i=1}^{N_{el}} C_{el}(el_i) \tag{3}$$

The cost of porting C_{port} of a process definition *p* is the sum of the element cost C_{el} for each element el_i from E. The element cost C_{el} for an element el_i of process definition *p* is defined as follows:

$$C_{el}(el_i) = \max_{j=1...N_{ta}} (V(ta_j, el_i) * D_{ta}(ta_j)) \quad (4)$$

$V(ta_j, el_i)$ denotes the testing function that returns 1 if el_i violates a test assertion ta_j and 0 otherwise. $D_{ta}(ta_j)$ denotes the degree of the test assertion ta_j (i.e., the number of engines that do not support the feature tested by a test assertion ta_j). This means if an element el_i does not violate ta_j (i.e., $V(ta_j, el_i) = 0$), the element cost for the combination of the two amounts to zero. If el_i violates the assertion, the element cost depends on the amount of engines that do not support the feature tested by the assertion. The more engines that support the feature, the less the cost of porting it will be. The lower the amount of engines, the higher the cost. The *max* function takes into account that a single element can violate multiple assertions. This can happen if there are multiple different problematic configurations of the element. An example of such an element in BPEL is the `reply` activity. The activity may be used to report a fault to a client of the process, by setting its `faultName` attribute and it may be linked to a `receive` activity by setting its `messageExchange` attribute. Both of these attributes are independent of each other, but not fully portable. We select the maximum of the degrees based on the assumption that the least portable part of the element will have the highest impact. This is a design decision and alternative schemes are possible. For instance, the degrees of all violations could be summed up and normalized. We tested several schemes for aggregating degrees when evaluating the metrics (cf. section V-C), but could not find significant differences in the metric values and therefore selected the simplest approach (i.e., the max function) here.

Summarizing the above discussion, the weighted elements metric M_e is calculated as follows:

$$M_e(p) = 1 - \frac{\sum_{i=1}^{N_{el}} \max_{j=1...N_{ta}} (V(ta_j, el_i) * D_{ta}(ta_j))}{N_{el} * N_{engines}} \quad (5)$$

C. Activity Portability Metric

The most central building block of process or workflow languages in general, and BPEL in particular, are activities. Activities are typically basic atomic steps of computation. In graph-based languages, they are connected in a process graph by defining control dependencies between them. In block-structured languages, composite activities group other activities and thereby define the control-flow [27]. BPEL provides both types of control-flow definition, either through direct control links or structured activities. In process complexity measures [22], [23], activities and the transitions among them are the dominant factor.

Apart from activities, process definitions include a variety of other elements such as variable definitions. Considering the conceptual importance of activities, it could 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 behaviour which is not desirable.

To provide an activity-oriented view on portability, we define the *activity* portability metric M_a 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 the portability metric. The elements that count as activities are defined as such in the BPEL specification [7, pp. 84–146]. Portability issues that are detected for subelements directly belonging to a specific activity, as for example for `copy` elements of an `assign` activity, are counted as issues for that activity. Issues that cannot be linked to a specific activity, as for example process-level `import` statements, are omitted in the consideration of this metric. For M_a, C_{new} changes to:

$$C_{new}(p) = N_a * N_{engines} \quad (6)$$

where N_a denotes the total amount of activities, the cardinality of A, in the process definition. C_{port} changes to:

$$C_{port}(p) = \sum_{i=1}^{N_a} C_{el}(a_i) \quad (7)$$

This means that only the element cost C_{el} of the activities in p is considered.

D. Service Communication Portability Metric

Communication and composition relations among services are a decisive factor for service-oriented systems and metrics for such systems center on these properties [17]. Communication relationships describe the *observable behaviour* of services; that is, the messages they send and receive. The distinction between the description of observable and internal behaviour is the discriminating factor for different types of service composition languages [28]. Composing services through message sending and reception is the main purpose of service orchestration languages, such as BPEL. This task is performed using the specific activities for sending, receiving and replying messages. 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 behaviour. However, this is unlikely if the activities that have to be changed, concern communication. In this case, these activities directly affect the observable behaviour of a process. Changing them (to enable portability) and consequently changing the observable behaviour influences other systems that interact with the process, which is generally undesirable.

The service communication portability metric M_s allows to view 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

Table I
SUMMARY OF PROPOSED METRICS

Portability Metric	Description
Basic, M_b	Relates number of problematic elements to total number of elements
Weighed Elements, M_e	Extends M_b by considering the degree of an element
Activity, M_a	Similar to M_e, but is limited to activities instead of elements
Service Communication, M_s	Similar to M_e, but is limited to activities for service communication instead of elements

service interaction (i.e., the elements of set S):

$$C_{new}(p) = N_s * N_{engines} \tag{8}$$

$$C_{port}(p) = \sum_{i=1}^{N_s} C_{el}(s_i) \tag{9}$$

Effectively, this is an extension of M_a that focuses solely on activities for service interaction. N_s 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.

Table I summarizes the different metrics proposed.

V. EVALUATION AND VALIDATION

Validation of new proposed metrics is crucial [29], both from the theoretical and from the practical point of view. A theoretical validation clarifies the properties of metrics and thus helps to determine when the metrics can be used in a meaningful way. An empirical validation demonstrates the applicability of the metrics and exemplifies their interpretation. In the next sections, we validate our metrics theoretically and empirically using different frameworks, following the methodology applied in similar studies [30].

A. Validating Construct Validity

We validate the metrics theoretically using two well-known validation frameworks [12], [13]. The first [13] focuses on *construct validity* and is to be used to define the scope of a metric. It specifies ten aspects that should be discussed for metrics to clarify their purpose and crucial characteristics, which we do in the following for all of our metrics in combination. For each aspect, typically a set of different properties is available and suggested by [13], one or more of which can be applicable for a metric. By the term *attribute*, [13] refer to the property that is to be measured, in our case portability. By the term *measurement instrument*, they refer to the tool used to compute metric values, in our case the static checker (cf. section III).

1) Purpose of the metrics: The metrics inform developers and system administrators about the portability characteristics of their software. When the change of a runtime system becomes necessary, they help to make a decision on whether to invest in porting or rewriting software.

2) Scope of the metrics: The metrics are of technical nature and are applicable during and after development. Their scope is typically a single (service and process-oriented) project of one workgroup.

3) Measured attribute: The metrics address a quality attribute of service- and process-oriented software. Specifically, they address its portability.

4) Natural scale of the attribute: Portability of software naturally ranges between two poles, full portability without a single modification and no portability of any part of the program. This resembles an interval scale.

5) Natural variability of the attribute: Being a technical attribute, portability is not subject to variations that are common for attributes involving human factors, such as the performance of a person depending on the time of the day. We can only determine the variability of the attribute by observing it in practice and considering the ranges in which in typically varies. Therefore, we defer this discussion to the empirical evaluation in section V-C.

6) Definition of the metrics: The metrics and the functions for computing them have been formally defined in sections IV-A to IV-D. The measurement instrument used is *counting* (i.e., counting of portability issues in code). The measurement is automated.

7) Natural scale of the metrics: The metrics have a rational scale, ranging in the interval between zero and one.

8) Natural variability of the instrument: This aspect refers to the measurement error of the metrics. Our metrics rely on empirical data of language support in engines. As a consequence, the main source of measurement error in the instrument stems from incomplete or faulty data. We did not benchmark all engines that exist for BPEL, which is hardly feasible due to the effort associated with benchmarking and the licensing cost and strategy of several engine vendors. Hence, the data is not fully complete and strongly increasing the number of engines may also lead to a differing output of the measurement instrument. Another source of measurement error could come from faults in the benchmark that indicate portability issues where there are none (false-positives) or do not discover certain issues (false-negatives). It is not possible to prove that no such errors exist. As the benchmark code is available to public scrutiny and has indeed been corrected by experts from other groups, we have confidence that this error rate is negligible.

9) Relationship between metrics and attribute: Our metrics are directly related to the measured attribute, portability. If for instance a non-standard extension element is introduced in the code, this will limit the overall portability of the process definition. The usage of this element will be detected by our measurement instrument and influence the metric values accordingly.

10) Natural side effects of using the instrument: As a general rule, the results of a measurement may change depending on the effort put into the measurement process

itself. Especially manual measurement is prone to this error, as for example more time spent in measuring might result in more desirable metric values without any change in the underlying attribute. As we have automated the measurement process fully, there is no room for human bias in the measurement instrument.

B. Validating Metric Properties

The second theoretical validation framework [12] is grounded in measurement theory and defines certain types of metrics as well as the mathematical properties that should be satisfied by each type of metric. The model underlying [12] is that of a system which contains elements, relations between elements, and modules. In our case, elements map to elements of the process, relations to their ordering in the process graph and modules to sets of connected elements. The metric types are *size*, *length*, *complexity*, *cohesion*, and *coupling* metrics. Although there is no direct fit of the metrics proposed here to this framework, it is beneficial to discuss what kind of properties our metrics fulfill. The metrics presented here are formed by the relation of two metrics C_{port} and C_{new}, with different ways of calculating them. These metrics are complexity metrics in the sense of [12]. This means they fulfill the properties of *non-negativity*, *null value*, *symmetry*, *additivity*, and *monotonicity*. The purpose of relating C_{port} and C_{new} is to obtain a normalization which enables the comparison of programs of different sizes concerning their portability. Hence, our metrics are *normalized complexity metrics* which do no longer fulfill all properties of classical complexity metrics. In the following, we discuss each of the different properties.

1) Non-negativity: Both, C_{port} and C_{new}, are obtained by adding up positive numbers, so they are always positive. From this follows that the property of non-negativity also applies to the normalized metrics:

$$M_{port}(p) \geq 0 \tag{10}$$

2) Null value: The null value property requires that the complexity of an empty system, or in our case program, must be null. For an empty program the cost values will be zero. As a consequence, the normalized metrics result in a division by zero and are undefined. Therefore, the metrics do not fulfill the property of null value, precisely because they are normalized. As a general rule, the application of the metrics on empty programs is not meaningful.

3) Symmetry: Symmetry for complexity metrics requires that the complexity value does not depend on the labeling used for the relationships between elements. In our case, the relationship between elements translates to their ordering in the process graph. This means that two process definitions p and p^{-1} with an identical set of elements E that have different orders $order$ and $order^{-1}$ with the same control-flow semantics, should have the same metric values $M_{port}(p)$ and $M_{port}(p^{-1})$. Reordering is possible for a variety of elements, for example when used for parallel processing or event handling. The cost metrics calculate the cost on a per-element basis, so the ordering is irrelevant and the metrics are symmetrical. As a consequence, also the normalized metrics are symmetrical. In the notation of [12]:

$$\begin{aligned}(p{=}<E, order> \wedge\, p^{-1}{=}<E, order^{-1}>)\\ \Rightarrow M_{port}(p) = M_{port}(p^{-1})\end{aligned} \tag{11}$$

4) Additivity: Additivity requires that the complexity of a program that is composed of two disjoint modules is equal to the sum of their complexity. Although this applies to C_{port} and C_{new}, summing up the normalized portability metrics is meaningless, due to normalization.

5) Monotonicity: Monotonicity requires that the complexity of a program is no less than the sum of the complexity of two unrelated parts of it. In our case this can be illustrated by two parallel branches, p^1 and p^2 of the program p. The complexity of the overall program should be at least as high as the sum of the complexity of the two branches. For C_{port} and C_{new}, this property clearly holds. However, this does not apply for the normalized metrics. Due to the normalization, the metric value $M_{port}(p)$ always is in the interval of $M_{port}(p^1)$ and $M_{port}(p^2)$ and not equal or higher than the sum of the two. Nevertheless, it is still monotonic. For instance, let $C_{port}(p^1) < C_{port}(p^2)$, $C_{new}(p^1) < C_{new}(p^2)$, and $M_{port}(p^1) < M_{port}(p^2)$. From the additivity of complexity metrics we get:

$$\begin{aligned}M_{port}(p) &= \frac{(C_{port}(p^1) + C_{port}(p^2))}{(C_{new}(p^1) + C_{new}(p^2))}\\ &> \frac{C_{port}(p^1)}{C_{new}(p^1)} = M_{port}(p^1)\end{aligned} \tag{12}$$

From this follows that $M_{port}(p) > M_{port}(p^1)$: The portability of the program will always be larger than the lower bound of the portability of two disjoint parts of it.

Summarizing the discussion, we can see that the metrics are normalized complexity metrics which fulfill the properties of non-negativity, symmetry, and monotonicity. They fail to satisfy the properties null value and additivity due to normalization.

C. Empirical Evaluation

For an empirical evaluation, we need data in the form of concrete process definitions for which the metrics can be computed. Here, we use four libraries of process definitions, three of them coming from different BPEL vendors, being ActiveEndpoints, Apache ODE and Oracle. These libraries are freely available and serve as documentation and tests for the respective engines. We obtained the fourth collection from a set of different and more heterogeneous sources where BPEL process definitions are publicly available, such as the BPELUnit mailing list, Stackoverflow, and a collection of workflow patterns in BPEL [31]. We pseudonymize the

libraries, because of licensing reasons for one of them. In total, the amount of process definitions adds up to a set of 215 process definitions. The processes vary strongly in complexity, ranging from eight to 168 elements, and cover all features of BPEL. The size of each library is listed in Table II and varies from 22 to 86.

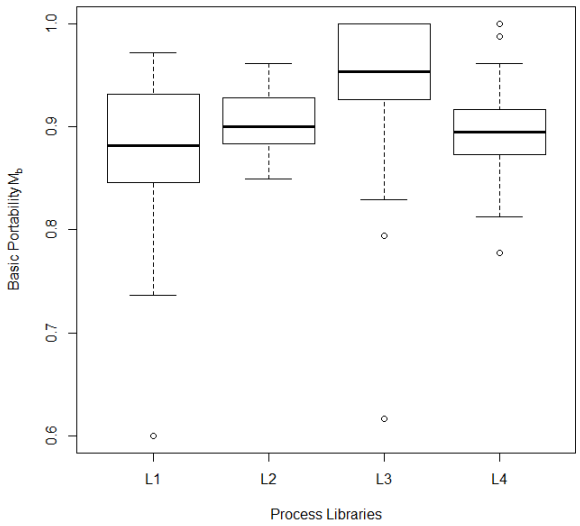

Figure 2. Basic Portability for Process Libraries

1) Natural Variability of Portability: Table II shows descriptive statistics for the process libraries with the different metrics. Additionally, Fig. 2 depicts boxplots for the different process libraries and their basic portability. For all metrics and process libraries, mean portability values are relatively high, ranging at values of 0.9. All standard deviations are relatively low, with the highest value of 0.13 for L_1 and M_b. This indicates that the processes of each library do not deviate strongly and, despite their differences in functionality, do share a common level of portability. At a first glance, this level may seem high. It is important to keep in mind, that the language and systems we consider here specifically aim to produce portable code. The question is what can be considered as high in this domain, or, in the terms of [13], what the natural variability of the attribute is here (cf. section V-A5). With the data at hand, we can provide a hint for the natural variability of the portability of service-oriented processes. Considering the aggregated means and standard deviations of the different metrics for all processes, listed in the last column of Table II, the data indicate that the portability naturally ranges at 0.94 and varies with a deviation of 0.07. Anything below this niveau can be considered as to be of lower quality.

In [10], even the top three engines in terms of successful conformance tests share only 45 % of the total test set. This implies that engines implement relatively disjoint sets of the language (cf. Fig. 1) and as a result lower portability values than the ones discussed here could be expected. However, disjointedness in language support only results in portability issues if processes use features that are not well supported. If they mainly use features of the basic language subset that is widely supported (i.e., that are in the set of 45 %), higher portability will be the result. The latter is the case for all of the four libraries.

2) Description of the Process Libraries: Looking at $L1$, relatively low values for M_b and M_e with values of 0.84 and 0.87 respectively, along with relatively high standard deviations, contrast high values for M_a and M_s. This reveals that the main portability issues do not lie in the control-flow and communication activities of the processes, these are indeed almost fully portable. The issues here reside mostly in the usage of non-standard extensions. Such issues, as they relate to extensions for logging, etc., tend to be fixable.

For $L2$, values of M_a and M_s are lower in total, and also lower than M_e. This indicates the opposite structure than for $L1$. Portability issues mainly originate from the activities and the control-flow definition. Whereas the process definitions do not make heavy use of language extensions, they rely on configurations that are of limited portability. Porting this process library will be comparatively harder.

$L3$ achieves high and similar values for all metrics. Low standard deviations along with a high number of processes provides confidence that the process library as a whole is of high quality. The only exception is M_s, where a comparably low portability value of 0.93 is combined with a relatively high standard deviation of 0.12. This shows that there are a few outliers with low portability to be found.

Finally, $L4$ shows similar values as $L1$, although with lower numbers in total. Portability issues mainly come from non-standard elements that do not directly relate to activities or communication aspects.

In total, $L3$ scores best when it comes to portability in general and a final decision is indicated by the aggregated metric values in the last column of Table II. This is also illustrated by Fig. 2, which shows the basic portability of the libraries, where $L3$ clearly ranges at the top. The other libraries lie at the border of an acceptable level, with $L1$ scoring lowest and showing the highest degree of variation. Focusing on the process and communication view (M_a and M_s), $L3$ is overtaken by all but $L2$.

3) Effect of Code Size: Especially complexity metrics are prone to a distorting effect due to code size [32]. Metrics tend to vary for systems of differing size and therefore are unsuitable for the comparison of arbitrary programs. It is a quality property of a metric to be resilient to changes in code size. To compare the effect of code size on our metrics, we extract two groups of processes, large and small processes, from all of the libraries. Large processes come from the fourth quartile in terms of the number of elements (i.e., the 25% largest processes) and small processes from the

Table II
DESCRIPTIVE STATISTICS FOR PROCESS LIBRARIES

Collection	N	Statistics	M_b	M_e	M_a	M_s	Ø
$L1$	22	Mean	0.84	0.87	0.99	0.99	0.92
		Std. Dev.	0.13	0.11	0.02	0.06	0.11
$L2$	25	Mean	0.90	0.97	0.92	0.94	0.93
		Std. Dev.	0.03	0.01	0.04	0.08	0.05
$L3$	82	Mean	0.95	0.98	0.96	0.93	0.95
		Std. Dev.	0.06	0.02	0.05	0.12	0.07
$L4$	86	Mean	0.90	0.93	0.96	0.97	0.94
		Std. Dev.	0.04	0.03	0.06	0.09	0.07
All	215	Mean	0.91	0.95	0.96	0.95	0.94
		Std. Dev.	0.07	0.06	0.05	0.10	0.07
Small Processes, Q_1	56	Mean	0.91	0.94	0.98	0.99	0.96
		Std. Dev.	0.11	0.09	0.03	0.03	0.08
Large Processes, Q_4	55	Mean	0.91	0.95	0.94	0.94	0.94
		Std. Dev.	0.06	0.04	0.07	0.11	0.07

Table III
R^2 FOR THE METRICS

	M_b	M_e	M_a	M_s
M_b		0.73	0.07	0.02
M_e			0.01	0.00
M_a				0.29

first quartile (i.e., the 25% smallest processes). Table II lists descriptive statistics for these sets in the last two rows. The mean values for large and small processes are very similar for all metrics and even identical for M_b. Differences in standard deviation are stronger, but only up to 0.08 which is still quite small. From this we can conclude that the effect of code size or changes to that size are negligible.

4) Information Carried by the Metrics: An important property of the different metrics is their ability to provide diverse information. This can be determined by looking at their correlation. If all metrics correlate strongly to each other, then strictly speaking they all carry similar information. If they all carry similar information, then there is no point in computing all of them. Instead, the simplest one is sufficient and the remaining ones can be discarded. The square of the linear correlation coefficient R^2 is suggested in [29] to evaluate correlation. Table III outlines R^2 for the different metrics. Except for M_b and M_e, all combinations show only a weak correlation. This means that they really provide different information and therefore it is beneficial to compute and consider them all. M_b and M_e show a strong correlation which can be attributed to the strong similarity in their computation. So, from an information-theoretic viewpoint, M_e is not superior to M_b. It is still beneficial to look at M_e, for reasons discussed in the following.

5) Discriminative Power: A central purpose of quality metrics is the ability to discriminate between different pieces of software. This ability is called the *discriminative power* of a metric. A metric that often assigns the same values to different pieces of software is not desirable, as it lacks this central property. Discriminative power can be measured by calculating the amount of unique metric values in the total set of values. The higher the amount of unique values, the better the metric is able to discriminate between different pieces of software. For M_b the discriminate power amounts to $103/215 = 0.48$, for M_e it is $133/215 = 0.62$, for M_a: $67/215 = 0.31$, and for M_s: $28/215 = 0.13$. Clearly, M_e has the highest degree of discriminative power. In combination with the fact that it correlates highly to M_b, we claim that for that reason, M_e is preferable to M_b and the specialized metric indeed does have an added value. Both, M_a and M_s, have a more limited degree of discriminative power which is expected as they abstract from certain aspects compared to M_b. However, as demonstrated in the previous paragraph, they do carry information different from M_b and M_e and highlight more critical portability issues.

VI. SUMMARY AND FUTURE WORK

In this paper, we presented a measurement framework for quantifying the portability of executable service-oriented processes, in particular BPEL process definitions. This framework provides an answer to the research question: *How can portability be measured for service-oriented processes?* The idea of using the amount of runtimes supporting certain language elements is an extension to classical portability metrics. Our measurement framework considers the portability of processes from different viewpoints that vary in their severity. We provide an extensive evaluation of the metrics, both from a theoretical and a practical angle. The theoretical validation confirms construct validity and shows that our metrics are normalized complexity metrics. The practical evaluation demonstrates their application.

One area of future work is the improvement of the measurement framework. A higher amount of engines included in the benchmark would enhance the quality of the metrics calculation. Especially commercial engines are of interest and the benchmark of several of these is ongoing work. Another area where work is needed is the quantification of further perspectives of the portability of service-oriented software. For instance, characteristics such as replaceability, installability or adaptability are defined as subcharacteristics

of portability [2] and should be measured as well. Replaceability could be incorporated in this framework by modifying the degree for elements that have multiple alternative representations in the language, such as looping structures.

References

[1] 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.

[2] ISO/IEC, *Systems and software engineering – System and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*, 2011, 25010:2011.

[3] B. Boehm, J. Brown, and M. Lipow, "Quantitive Evaluation of Software Quality," in *2nd Proceedings of ICSE*, San Francisco, USA, October 1976.

[4] T. Gilb, *Principles of Software Engineering Management*. Addison Wesley, 1988, ISBN-13: 978-0201192469.

[5] G. Athanasopoulos, A. Tsalgatidou, and M. Pantazoglou, "Interoperability among Heterogeneous Services," in *IEEE SCC*, Chicago, USA, September 2006.

[6] OASIS, *Topology and Orchestration Specification for Cloud Applications*, March 2013, v1.0.

[7] ——, *Web Services Business Process Execution Language*, April 2007, v2.0.

[8] OMG, *Business Process Model and Notation (BPMN)*, January 2011, v2.0.

[9] WfMC, *Process Definition Interface – XML Process Definition Language*, August 2012, v2.2.

[10] S. Harrer, J. Lenhard, and G. Wirtz, "BPEL Conformance in Open Source Engines," in *IEEE SOCA*, Taipei, Taiwan, December 2012.

[11] M. Glinz, "A Risk-Based, Value-Oriented Approach to Quality Requirements," *IEEE Computer*, vol. 25, no. 8, pp. 34–41, 2008.

[12] 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.

[13] 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.

[14] M. Geiger and G. Wirtz, "Detecting Interoperability and Correctness Issues in BPMN 2.0 Process Models," in *ZEUS, Rostock, Germany*, February 2013.

[15] L. Cesari, A. Lapadula, R. Pugliese, and F. Tiezzi, "A tool for rapid development of WS-BPEL applications," in *SAC, Sierre, Switzerland*, March 2010.

[16] T. Hallwyl, F. Henglein, and T. Hildebrandt, "A standard-driven implementation of WS-BPEL 2.0," in *SAC, Sierre, Switzerland*, March 2010.

[17] H. Hofmeister and G. Wirtz, "Supporting Service-Oriented Design with Metrics," in *IEEE EDOC*, Munich, Germany, September 2008.

[18] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling Metrics for Predicting Maintainability in Service-Oriented Designs," in *IEEE ASWEC*, April 2007.

[19] D. Bianculli, W. Binder, and M. L. Drago, "Automated Performance Assessment for Service-Oriented Middleware: a Case Study on BPEL engines," in *Int. Conf. on World Wide Web*, Raleigh, North Carolina, USA, April 2010, pp. 141–150.

[20] Y. Wang, Y. Taher, and W.-J. van den Heuvel, "Towards Smart Service Networks: An Interdisciplinary Service Assessment Metrics," in *IEEE EDOC Workshops*, September 2012.

[21] 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, 2010.

[22] I. Vanderfeesten, J. Cardoso, J. Mendling, H. Reijers, and W. van der Aalst, *Quality Metrics for Business Process Models*. Future Strategies, May 2007.

[23] J. Cardoso, "Business Process Quality Metrics: Log-Based Complexity of Workflow Patterns," in *OTM CoopIS*. Springer-Verlag, 2007, pp. 427–434.

[24] G. Muketha, A. Ghani, M. Selamat, and R. Atan, "Complexity Metrics for Executable Business Processes," *Information Technology Journal*, vol. 9, no. 7, pp. 1317–1326, 2010.

[25] ISO/IEC, *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality*, 2013, 25023.

[26] S. Harrer and J. Lenhard, "Betsy – A BPEL Engine Test System," University of Bamberg, Bamberger Beiträge zur WI und AI no. 90, July 2012, technical report.

[27] 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, GI*, vol. 4, no. 1, pp. 3–13, 2009.

[28] C. Peltz, "Web Services Orchestration and Choreography," *IEEE Computer*, vol. 36, no. 10, pp. 46–52, October 2003.

[29] IEEE, *IEEE Std 1061-1998 (R2009), IEEE Standard for a Software Quality Metrics Methodology*, 1998, revision of IEEE Std 1061-1992.

[30] D. Basci and S. Misra, "Measuring and Evaluating a Design Complexity Metric for XML Schema Documents," *Journal of Information Science and Engineering*, vol. 25, no. 5, pp. 1405–1425, 2009.

[31] J. Lenhard, A. Schönberger, and G. Wirtz, "Edit Distance-Based Pattern Support Assessment of Orchestration Languages," in *OTM CoopIS*, Hersonissos, Crete, October 2011.

[32] K. E. Emam, S. Benlarbi, N. Goel, and S. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, 2001.