Measuring the Installability of Service Orchestrations Using the SQuaRE Method

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

***Abstract*—Service-oriented software consists of middleware, such as application servers and runtime engines, into which service applications are deployed. This middleware is often complex and difficult to install. The deployment of services requires the crafting of deployment descriptors and packaging of applications. As a consequence, the installation of service-oriented software systems can be a daunting task. *Installability*, however, is an important influencer of the portability of software. Portability in turn is one of the main goals of service orchestration languages based on open standards. In this paper, we investigate the installability of service orchestrations based on the *Systems and software Quality Requirements and Evaluation* (SQuaRE) method, the new series of software quality standards currently under development by the ISO/IEC. We develop a measurement framework based on SQuaRE and tailored to evaluating the installability of service orchestrations and their runtimes. We validate the measurement framework theoretically and show its applicability in a case study.**

***Keywords*-SOA, SQuaRE, installability, metrics**

I. Introduction

Service-oriented software systems are by definition client-server systems [1]. One part of these systems, the service, runs on server-sided middleware, such as an application server. Application servers have long been used to provide crosscutting concerns to the applications they host, while also providing a balanced level of quality of service and scaling. Especially at the enterprise-level, such software packages are very complex. As a consequence of this complexity, they are costly and installing them into the runtime systems of an enterprise can constitute a considerable effort.

This cost in setup contradicts several of the goals of service-oriented architectures; that is, intrinsic interoperability of heterogeneous systems and flexibility through the portability of applications [2,3]. This applies in particular to service orchestrations, which are frequently implemented in process languages that build on open standards, such as the Web Services Business Process Execution Language (BPEL) [4] or the Business Process Model and Notation (BPMN) [5]. For these standards, the portability of process definitions is a fundamental goal. A high degree of portability and, as a consequence, an easy installation is emphasized by the advent of cloud-based services. The pricing models of cloud environments increase the importance of the ability to easily switch a runtime environment. In current environments, however, services themselves must be tailored to run on a cloud-platform through vendor-specific deployment information [6]. This effectively locks-in cloud users to a cloud vendor. One step towards fixing these issues of installability is their investigation and quantification through metrics which we address here. This can be useful for the service-oriented computing community, as here flexibility and hence portability, for which installability is a prerequisite, is of particular interest [2,3].

Portability is also a top-level quality characteristic in several software quality models [7]–[9]. Currently, the ISO/IEC is working on the *Systems and software Quality Requirements and Evaluation* (SQuaRE) method [8], a major revision of its series of quality standards. In this context, the ISO/IEC is redefining the metrics framework for measuring characteristics such as portability or installability and designated standards [10] are still under development. In [8], *installability* is defined as a subcharacteristic of portability, and should therefore be measurable. In this paper, we measure the installability of a type of software where portability is a central goal, being service orchestrations. In particular, we are trying to answer the question:

How can the installability of service orchestrations and their runtimes be measured?

To answer this question, we build a metrics framework aligned with SQuaRE that takes into account the installability of service orchestrations from two viewpoints and combines these for an overall judgment of installability. We rely on existing metrics from previous quality standards [11,12], apply them when reasonable, and suggest new metrics suitable for this domain. More specifically, we also take into account the effort of preparing applications for deployment and putting them into production. We validate the metrics theoretically using two validation frameworks [13,14] and use them to evaluate a set of BPEL processes and engines, thus demonstrating the applicability of the metrics.

In the next section, we outline related work, followed by the formal definition of our metrics framework. Thereafter, we present the theoretical validation and practical evaluation of the framework. Finally, we conclude the paper with a summary and an outlook on future work.

II. Basics and Related Work

In the following sections, we discuss (A) the underlying quality model (B) work on metrics for service orchestrations, and (C) alternative approaches for measuring installability and deployability.

A. Quality Models for Installability

The ISO/IEC family of software quality standards builds on several quality models [7,9] and is the basis for many quality evaluations today. It is currently being revised in the context of the SQuaRE method [8], which also forms the basis of the quality evaluation in this paper. The quality model defines eight top-level quality characteristics of software, each of which subdivides into several subcharacteristics that can be measured independently. In this paper, the subcharacteristic *installability* of the top-level characteristic *portability* is of relevance. At the time of writing, the specification that is to contain concrete metrics [10] is still under development and not yet open to public scrutiny. Hence, we consider the metrics from preceding versions of the ISO/IEC quality standards [11,12]. The quality model is rather general and tailored to stand-alone software. Here, we discuss the suitability of the metrics for service orchestrations and extend the framework with a measurement of *deployability*. We do not focus on metrics based on the observation of human behaviour, which contrasts several of the metrics proposed in [11,12]. Instead, we rely on metrics that can be computed automatically or through code inspection, which bears a benefit of cost efficiency and repeatability.

B. Work on Metrics for Service Orchestrations

Quality metrics for process models, which are also largely applicable to service orchestrations, mostly build on classical concepts such as coupling and cohesion [15] or focus on process complexity [16]. When it comes to portability or installability, relatively little work is available. In previous work [17], we examine the *direct portability* of process code among different runtime environments (i.e., the ability to directly port code, without the need for replacement or adaptation). A consideration of the installability of process runtimes and applications, like the one in this paper, is orthogonal to this. Even if process definitions can be directly ported, the new runtime environment has to be installed.

C. Alternative Measurement Approaches

Installability can also be viewed as the question of whether a set of applications can be installed next to each other on the same machine [18]. 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 property is also covered in the SQuaRE method, but there it is denoted as *co-existence* [8, p. 11]. Installability, on the other hand, refers to the cost or effort for the installation, given an installation is possible to begin with.

Deployability of software can also be considered as the cost 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 [19]. Our point of view on deployability is different here. We do not consider the network-wide deployment of a service, but instead the cost of deploying an application on a single host. This view is orthogonal to a network-wide deployment and our framework could be combined with such an approach.

An alternative to automated measurement that has also been used for measuring installability, deployability, and, in particular, usability in different domains is a *heuristic evaluation* or *cognitive walkthrough* [20]. In these methods, a user steps through a procedure, such as an installation, and judges its appropriateness for the task at hand. These techniques are especially useful for the evaluation of user interfaces [20]. In [21], they are used to analyze the installability and deployability of an application for anonymous web browsing. Here, we also use heuristic evaluation to quantify the complexity of an installation procedure, but we evaluate installation scripts instead of user interfaces.

III. Measuring Installability

We derive the measurement framework based on a case study of a set of BPEL engines and processes running on these engines. That way, all metrics are motivated practically and we can ensure their applicability. In this section, we define and explain the framework. A closer description of the engines and processes follows in the context of the practical evaluation in section IV-B.

Installability is defined as the "*degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment*" [8, p. 15]. When it comes to quantifying installability, only few metrics are available and in [11] all of them are marked as experimental. Furthermore, several metrics effectively measure the same thing. For instance, the metrics *effortless installation*, *installation ease*, and *ease of users manual installation operation* [11, p. 64] all measure the extent to which user actions are needed during installation. This translates to the notion of *installation effort* here. The metric *operational installation effort reduction* [11, p. 64] is used to measure effort reduction in the case of procedural changes. As we do not focus on procedural changes, the metric is of no relevance here. *Installation flexibility* [12, p. 37] relates the number of customizations implemented for the installation process, such as installation paths or port numbers, to the number of customizations required. The larger the extent to which customizations can be implemented, the better. In

our case study (cf. section IV-B), we could implement all customizations needed in all cases. As a consequence, this metric does not bear a benefit here, and thus we exclude it from our framework. The remaining metrics relevant to the evaluation here are *installation effort* and *ease of setup retry* [12, p. 37]. In the following, we discuss the applicability of these metrics, and define new ones.

A. Overview of the Measurement Framework

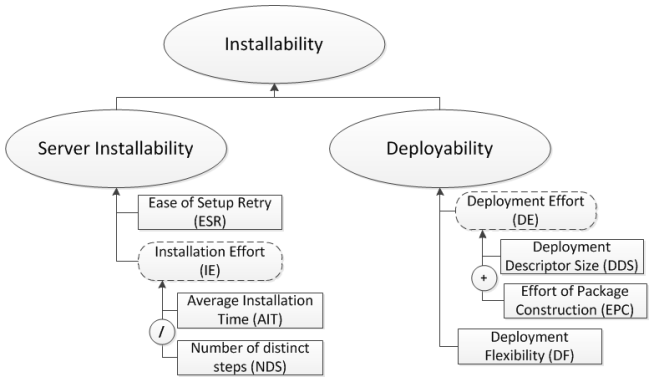

Figure 1. The framework for measuring installability. Ellipses denote quality attributes to be measured, rectangles denote direct metrics obtained through code analysis and benchmarking, and rounded dashed rectangles depict aggregated metrics that are computed by the combination of direct metrics using the functions displayed in circles.

Fig. 1 outlines the model we use for measuring installability. The quality attribute *installability* is subdivided into the subattributes *server installability* and *deployability*. This distinction is necessary as the focus here is not a standalone software product, but a service orchestration. Each of the subattributes 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 [11,12]. We extended installation effort to also consider *average time complexity* (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 [11,12], so we develop a new set. This set consists 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) [8, p. 27].

B. Measuring Server Installability

Ease of setup retry (ESR) is intended to measure how easy it is to successfully repeat an installation [12, p. 37]. It relates the number of successful installations of the same piece of software s (N_{succ}) to the number of attempted installations in total (N_{total}). That is:

$$ESR(s) = \frac{N_{succ}}{N_{total}} \tag{1}$$

[12] 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 $ESR(s)$ equal to one. If it is not free of bugs, installations may fail, resulting in a lower ESR value.

Installation effort (IE) is intended to provide a notion of the difficulty of the installation process. [12, p. 37] suggests to measure it as the relation of automatable installation steps in relation to the total amount of prescribed steps. In our case, the server installation process can be automated fully, so such a definition would, like installation flexibility, be of little help. Still, the servers in the case study do differ in the amount of steps they require for the installation. What is more, they do require a vastly different amount of time for the installation, which can vary in orders of magnitude when comparing certain servers. For that reason, we deviate in the measurement of installation effort from [12] 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 [12]. It further corresponds to a *quality measure element* from the SQuaRE series [22, p. 21]. 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, and can be determined by a heuristic evaluation. In the case study, we took the server distributions as provided by a vendor and automated the process of setting up the distribution in our environment. The heuristic evaluation counts each step in our installation script. An example of these steps looks as follows:

1) Create or clean the installation directory
2) Unpack the distribution to the installation directory
3) Unpack the server core
4) Copy the BPEL runtime to the server core
5) Copy the SOAP runtime to the server core

The average installation time can be computed by performing the distinct steps required, identified by NDS, a suitable amount of times and measuring execution times. This is an *effort quality measure element* [22, p. 14]. AIT and NDS can be aggregated to a notion of installation effort (IE) per installation step:

$$IE(s) = \begin{cases} 0 & \text{if } NDS = 0 \\ \frac{AIT(s)}{NDS(s)} & otherwise \end{cases} \tag{2}$$

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.

C. Measuring Deployability

Deployability describes the effort required to put a service in its production environment. There is no direct representation or corresponding metrics for this attribute in SQuaRE. We derive new metrics from existing general-purpose quality measure elements from SQuaRE [22] as far as applicable.

Service deployment normally consists of the execution of a single server operation provided with the service executable. Nevertheless, deployment can take different forms, multiple of which can be supported by a server. 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 [12, p. 37] to this context. Typically, three different options are available:

1) a copy operation of a deployment archive into a specific directory, denoted as hot deployment,
2) the invocation of a deployment script, or web service,
3) a manual user operation using a GUI or web interface.

To be able to use one of the deployment operations for a service application, this application must be prepared for deployment. This normally requires the packaging of the service 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 service in a specific environment. 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 based on the *number of steps* quality measure element [22, p. 21] in a similar fashion as NDS. This means 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:

$$EPC(service) = N_{fc} + N_{dc} + N_{co} \tag{3}$$

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. In our case study, a very simple structure consists of a process file, an interface definition file, and a deployment descriptor file in one directory that is compressed. Decisive for EPC are, the deployment folder, the descriptor file and the compression operation, so $EPC = 1+1+1 = 3$. However, the structure can be vastly more complex and depend on various nested archives with multiple descriptors.

The deployment descriptor size for a service corresponds to the added size of all descriptor files needed:

$$DDS(service) = \sum_{i=1}^{N_{desc}} size(dd_i) \tag{4}$$

DDS is the sum of the size of all relevant descriptors $< dd_1, ..., dd_{N_{desc}} >$. In our case, two different types of descriptor files exist: i) Plain text files and i) 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. Here, we use two simple mechanisms to compute file sizes. For plain text files, a lines of code metric is appropriate. For the descriptors at hand, every non-empty and non-comment 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, including simple content, and attributes, excluding namespace definitions, N_{ea}, 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:

$$size(desc) = \begin{cases} LOC(desc), & \text{if } plain(desc) \\ N_{ea}, & \text{if } xml(desc) \end{cases} \tag{5}$$

Listing 1 outlines a simple descriptor file for a single service. The descriptor consists of four elements and four non-namespace attributes that are set, so the total size is eight.

Listing 1. Example of a Simple Deployment Descriptor

```
<deploy xmlns="..." xmlns:bpel="...">
   <process xmlns:tns="..." name="tns:SimpleService">
      <provide partnerLink="SimplePartnerLink">
         <service name="tns:SimpleServiceInterface"
              port="SimplePort"/>
      </provide>
   </process>
</deploy>
```

These two metrics can be aggregated to a combined version for deployment effort (DE), by adding them up:

$$DE(service) = DDS(service) + EPC(service) \tag{6}$$

The idea here is to capture every factor, independent of its nature, that increases the effort of deploying a service.

IV. Validation and Evaluation

Validation of software metrics is crucial to ensure that they measure what they are intended to measure and to clarify how they can be used in a meaningful way [23]. We have implemented a prototypic tool[1] that automates the

[1]See www.uni-bamberg.de/pi/port-metrics for more information.

computation of all metrics, except for NDS and DF, which are determined by heuristic evaluation. Since these metrics need to be computed only once per server environment, this effort should be acceptable. The tool computes the metrics through static code analysis or parsing of events in log files.

A. Theoretical Validation

The theoretical validation frameworks we use clarify the mathematical properties of the metrics on the one hand [13] and examine construct validity on the other hand [14].

1) Measurement-theoretic Validation: [13] proposes a framework for the axiomatic validation of structural software metrics. The authors list different categories of structural metrics and define a set of mathematical properties each type should fulfill. In this case only the internal metrics relating to descriptors and package sizes DDS, EPC, and DE are structural code metrics, so only they directly fit in this framework. Nevertheless, a clarification of the measurement-theoretic properties of all metrics is important. For this reason, we discuss the central properties from [13] for all metrics. The internal metrics are size metrics [13], which should fulfill *non-negativity*, *null value*, and *additivity*. An additional property required by nearly all categories in [13], which we also consider here, is *monotonicity*.

Non-negativity: This property applies to all metrics in the framework. Nearly all of the metric values are obtained by counting occurrences, being either operation steps (NDS), seconds elapsed (AIT), successful installations (ESR), available deployment options (DF), or elements and lines of code (DDS and EPC). As IE and DE are aggregated from two non-negative metrics, they are also non-negative.

Null value: Metric values for an empty system or program should be null. DDS and EPC will be zero for an empty program, as no descriptors or packages need to be built. Being the sum of the two, this also applies to DE. If there is nothing to install, NDS and DF will be zero as no steps are required or options available. In this case, IE is defined to be zero. Finally, if nothing is executed, the time elapsed will be zero (AIT) and nothing will successfully be installed (ESR), no matter how many attempts.

Additivity: Size metrics should be additive, meaning that the size of two disjunct systems taken together should be identical to the sum of the two. This property holds for DDS and EPC. Given two services s_1 and s_2 are separately packaged, the sizes of their descriptors and packages is completely independent. Hence, if they are deployed on the same server, forming a system s together, the values for DDS and EPC of that system will be equal to the sum of the values of the two services. As a consequence, this also applies to DE. Additivity does also hold for NDS: Given two servers are installed, all installation operations need to be completed for both of them. However, it does not hold for the remaining metrics. AIT, ESR, and IE are average values or aggregated thereof, so adding them up is meaningless. Also the number of deployment options (DF) does not necessarily increase with the number of servers.

Monotonicity: Metric values for the combination of two unrelated systems should be at least as high as the values for each of them. According to [13], this is no strict requirement for a size metric, but monotonicity in the above sense is implied from additivity. What is more, monotonicity also holds for AIT, ESR, IE, and DF. The time elapsed cannot decrease by installing more systems (AIT) and, similarly, a partial failure of an installation does still count as a failure (ESR). Moreover, adding more servers does not decrease the amount of deployment options available (DF).

2) Evaluating Construct Validity: The second theoretical validation framework [14] addresses *construct validity*. It is used to assess whether a metric really measures what it is intended to measure. The framework is qualitative and takes the form of ten aspects that should be clarified for each metric to define its scope and meaningful areas of application. By *attribute*, [14] refers to the aspect to be measured, in our case installability, and by *measurement instrument* the authors denote the mechanism with which the metrics are computed, in our case the tool.

Purpose of the metrics: The purpose of all metrics is the evaluation of the installability characteristics of service orchestrations and their runtimes. They can be used for private self-assessment of a workgroup or to inform third parties, such as customers and maintainers.

Scope of the metrics: The scope of the metrics is typically a single project from one workgroup that consists of multiple service orchestrations and runtimes.

Measured attribute: The attribute to be measured is the installability of the software, the ease with which it can be installed in a runtime environment. The installation effort metrics (NDS, AIT, and IE) measure the complexity of the installation process and ESR the reliability of the installation. The deployment effort metrics (DDS, EPC, and DE) measure the size of deployment artifacts and DF the flexibility of the deployment process.

Natural scale of the attribute: The natural scale of the attribute is independent on any metric that tries to quantify it. We have no knowledge on the natural scale of installability per se, but it is reasonable to assume that software products differ in their installability in a way that allows for an ordering. This implies that installability can be observed at least on an ordinal, and possibly on a rational scale [22].

Natural variability of the attribute: We have no knowledge on the natural variability of installability. However, it can reasonably be expected that installability varies depending on the environment into which a software is installed or the size of the system to be installed. This claim is also supported by the practical evaluation in section IV-B.

Definition of the metrics and the instrument: All metrics are formally defined in section III. They are computed by *counting* elements of code, *matching* installation steps or

product functions, and *timing* task executions [14, p. 4].

Natural scale of the metrics: Ease of setup retry (ESR) is measured on an interval scale of $[0; 1]$. All other metrics are measured on a ratio scale [22, p. 36]

Natural variability of the measurement instrument: This aspect refers to possible measurement errors in the metrics computation. Human judgement always involves a margin of error, so metrics determined by heuristic evaluation (NDS and DF) can yield inaccuracies. The computation of the remaining metrics is automated in a prototype tool, so we can at least guarantee reproducibility of the computation. The number of failed installations (ESR) and the time elapsed during installation (AIT and IE) likely depends on physical constraints such as the number of processors or memory available, and will be different on different machines. However, this is rather an inherent natural variability and not a computational error. We compute the descriptor size metrics (EPC, DDS, and DE) based on white-listing of relevant descriptor files. In case we omitted a file type, there is an error in the measurement instrument.

Relationship of the attribute to metrics values: All of the metrics are direct in the sense of [23]; that is, changes to the underlying attributes are directly reflected in the metrics. For instance, if more installations fail, ESR increases. If the installation procedure gets more complex, it will likely involve more steps and / or take longer (NDS, AIT, and IE). If deployment gets more complex, it will likely involve more steps to construct deployment archives and require larger descriptors (EPC, DDS, and DE). If a new option for deployment is available, DF increases.

Side-effects of using the metrics: Measurement of human behaviour is prone to side-effects, as humans could adapt their behaviour to produce desirable metric values without changing the underlying attribute. Here, we measure code artifacts, so there is no room for this type of error.

B. Experimental Results

An experimental evaluation of the metrics framework is important to verify its applicability and demonstrate its interpretation. We use a set of BPEL engines as a case study and evaluate their installability and the deployability of a set of functionally identical processes deployed on different engines. We also evaluate several third-party processes, running on specific engines. It should be noted that, although we focus on orchestration engines, the metrics framework should be applicable to a larger variety of environments, such as application servers in general. However, this claim should be supported by additional experiments that could also be used to confirm the usefulness of the framework. We defer such additional experiments to future work.

The runtimes available are six free and open source engines, the OpenESB BPEL Service Engine v2.3, Petals ESB Easy BPEL 4.1, Apache ODE 1.3.5, bpel-g 5.3, Orchestra 4.9 and ActiveBPEL 5.0.2. The first two are ESB solutions that include an orchestration engine, whereas the latter four are pure engines running in a servlet container. We modified the betsy tool, which we used in previous work to benchmark standard conformance of these engines [24,25], to gather data on the installation processes. The tool automatically deploys and executes conformance test cases on these engines. Also the installation of engines is automated in the tool. To be able to gather the data needed for the computation of metrics like AIT, we had to modify the installation process to print timing data in a suitable format into the log files, so that we could parse these files later. Furthermore, we analyzed installation scripts to compute metrics such as NDS. Finally, our prototypic tool for metric computation inspects the log files to compute the relevant metrics. The main benefit of extending betsy for this task is that the installation process of engines is automated and thereby reproducible, and works in the same fashion for all engines. This similarity enables a reasonable direct comparison of different engines, the lack of which is a common drawback in software comparisons [26].

1) Server Installability: Tab. I lists the metrics that characterise server installability in its upper half. To gather the data needed for the calculation of the metrics, we repeated the installation process of each engine 150 times, including a warm-up phase, on a machine running Windows 7, 64bit with an i7 quadcore processor, 16 GB of RAM, and a 1 TB SATA drive with 7200 RPM. These hardware requirements are far above the requirements specified for any of the engines. Thereafter, we mined the log files of these runs and analyzed the installation scripts implemented. The installation process for all engines consists of the setup of a core server environment, into which the engine needs to be copied or installed with a vendor-provided installation script, along with the setting of environment-specific configurations.

Several observations can be made based on the data here. Most engines require a similar amount of steps for installation (i.e., have a similar value for NDS). All but the engine with the lowest amount of steps, Petals, have a fully deterministic installation process (i.e., after one installation attempt, it is always possible to deploy and execute processes on the engine). For Petals, every tenth installation is a failure. In this case, the engine signals a successful installation, but certain components are missing which results in failures during later operation. We were unable to ultimately determine the reasons for these installation failures. Finally, for the average installation time, the engines differ strongly with up to two orders of magnitude. OpenESB forms an outlier with a very high installation time. This is due to one step, where a vendor-provided installation script is called which consists of a number of uncompression operations that take very long. The same applies for Orchestra and ActiveBPEL, although to a lesser degree.

Table I
INSTALLABILITY METRICS – SEPARATED BY SERVER INSTALLABILITY AND DEPLOYABILITY

Metric	OpenESB v2.3	Petals 4.1	ODE 1.3.5	bpel-g 5.3	Orchestra 4.9	ActiveBPEL 5.0.2
Server Installability of Engines ($N = 150$ runs)						
Number of distinct steps, NDS	7	5	6	6	7	6
Average installation time, AIT [sec]	133.88	3.13	3.31	3.01	42.53	22.91
Coeff. of variance of instal. time, CV_{time}	0.02	0.11	0.51	0.37	0.22	0.06
Installation effort, IE	19.13	0.63	0.55	0.50	6.08	3.82
Ease of setup retry, ESR	1	0.90	1	1	1	1
Deployability of Functionally Identical Processes ($N = 36$)						
Deployment flexibility, DF	2	2	3	2	2	2
Descriptor size, DDS (mean / st.d.)	73.92 / 7.37	78.5 / 6.31	10.69 / 1.75	9.11 / 2.81	0 / 0	21.36 / 5.96
Effort of package constr., EPC (mean)	14	9	2	2	1	5
Deployment effort, DE (mean / st.d.)	87.92 / 7.37	87.5 / 6.31	12.69 / 1.75	11.11 / 2.81	1 / 0	26.36 / 5.96

Being a mean value, AIT is vulnerable to outliers in the data. Given there is a high deviation in the data, AIT would not allow for a meaningful interpretation. To determine whether this is the case here, we computed the *coefficient of variation* (CV_{time}), which describes the relation between the mean value and the standard deviation of a variable. If the value of CV is larger than one, the underlying distribution is considered as having a high variation, otherwise it is considered as having a low variation. Low values apply for all our observations of AIT, which means that this metric indeed allows for a meaningful interpretation on its own. Finally, IE provides a way to directly view and compare the associated effort, and would, for instance, allow for a ranking of the different engines.

2) Application Deployability: In its second half, Tab. I lists deployability metrics for a set of processes, aggregated by the engines. These are 36 processes from [25] that can be deployed on all engines. This way, a direct comparison of the deployability of an engine is possible. To get an overall view of the sets of processes, we present the means and standard deviations. For EPC, the mean alone allows for a meaningful interpretation. As all packages are built in a very similar fashion here, standard deviations for EPC are zero for all cases. We also computed deployability metrics for the process libraries of OpenESB, ODE, bpel-g, and ActiveBPEL that serve as samples for the respective engine, and which tend to be more complex. Although we omit a closer presentation of all of these libraries for reasons of space, we can say that the libraries have EPC values similar, though slightly higher, to the ones in Tab. I. The values of all metrics vary strongly for the different runtime systems. For instance, OpenESB and Petals require descriptor sizes of more than 70 elements in the mean. For the other engines, this is significantly lower. One special case is formed by Orchestra. There, no deployment descriptors are required. All information needed is read from the source files, such as WSDL definitions, directly. Due to the self-descriptiveness of Web Services artifacts, such a strategy is possible and Orchestra demonstrates that it is feasible. Nevertheless, only few runtime systems make use of this. When looking at EPC values, it can be seen that the runtimes that require the largest deployment descriptors are also the ones that require the most complex archives (i.e., several nested archives containing zips and wars). For all other runtimes, archives are simply necessary for grouping all relevant files together, so that they can be deployed by linking a single file. DE aggregates the deployment effort for a direct comparison that allows for a ranking of the runtimes. Finally, when looking at the number of deployment options available (DF), it can be seen that all runtimes offer a similar level of flexibility.

3) Ranking the Servers: Summarizing the results, it can be said that both, ODE and bpel-g, provide a balanced level of good values for the metrics, with bpel-g being slightly ahead. Although it excels in terms of deployability, Orchestra takes somewhat longer to install (AIT and IE) and therefore ranks third. Even though ActiveBPEL is quicker than Orchestra in installation, the deployment is much more complicated (DE), leading to rank four. Finally, OpenESB and Petals both have relatively complex files with large descriptors (DDS and DE). OpenESB takes long to install (AIT and IE) and the installation of Petals is not stable (ESR).

4) Process Complexity and Deployability: So far we mainly used the metrics to compare the quality of runtimes, but their purpose is to compare single orchestrations, too. For that reason, we examine the processes from the ActiveBPEL library, a freely available set of processes of varying complexity, in Tab. II. This lets us observe the effects of process complexity on deployability. A basic metric for expressing the complexity of a service orchestration is the *number of services* [27] involved in it, independent of whether the services are implemented or just used by the orchestration. The processes in the library involve either one, two, or five services, and Tab. II depicts deployability metrics for these groups. EPC values are almost constant, although slightly higher due to the package structure typical in the library. Moreover, they are unaffected by the growth in complexity. Descriptor sizes on the other hand grow almost linearly to the number of services in the system. This demonstrates that an orchestration with many services of possibly fine granularity will be comparably harder to

Table II
DEPLOYABILITY OF PROCESSES OF DIFFERENT COMPLEXITY

$N_{services}$	$N_{processes}$	$\emptyset DDS$	$\emptyset EPC$	$\emptyset DE$
1	11	31.36	9	40.36
2	3	43.67	9.33	53.0
5	5	105.0	9	114.0

deploy than one with a few services of lower granularity.

V. CONCLUSION AND FUTURE WORK

In this paper, we have built a comparison framework for the installability of service orchestrations and their runtimes, thus answering the research question posed in the introduction. The framework is derived from the quality model and metrics definition of the SQuaRE method and extends it with metrics for deployability. We evaluated the metrics theoretically and demonstrated their practical usage. Our long-term goal is to construct a measurement framework for evaluating the portability of service-oriented and process-aware applications. Installability is one subcharacteristic of portability, but properties such as replaceability and adaptability need to be measured as well.

REFERENCES

[1] W3C, *Web Services Architecture*, February 2004.

[2] SOA Manifesto Working Group, "SOA Manifesto," in *SOA Symposium*, Rotterdam, The Netherlands, October 2009.

[3] R. Khalaf, A. Keller, and F. Leymann, "Business processes for Web Services: Principles and applications," *IBM Systems Journal*, vol. 45, no. 2, pp. 425–446, 2006.

[4] OASIS, *Web Services Business Process Execution Language*, April 2007, v2.0.

[5] OMG, *Business Process Model and Notation (BPMN) Version 2.0*, January 2011.

[6] 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.

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

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

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

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

[11] ——, *Software engineering – Product quality – Part 2: External metrics*, 2003, 9126-2:2003.

[12] ——, *Software engineering – Product quality – Part 3: Internal metrics*, 2003, 9126-3:2003.

[13] 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.

[14] C. Kaner and W. Bond, "Software Engineering Metrics: What Do They Measure and How Do We Know?" in *IEEE METRICS*, Chicago, USA, September 2004.

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

[16] 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.

[17] J. Lenhard and G. Wirtz, "Measuring the Portability of Executable Service-Oriented Processes," in *IEEE EDOC*, Vancouver, Canada, September 2013.

[18] R. D. Cosmo and J. Vouillon, "On Software Component Co-Installability," in *ACM SIGSOFT*, Szeged, Hungary, September 2011.

[19] IETF, *Metrics for the Evaluation of Congestion Control Mechanisms*, 2008, IETF Network Working Group.

[20] J. Nielsen, *Usability Inspection Methods*, 1994, Wiley, New York, ISBN: 978-0471018773.

[21] J. Clark, P. van Oorschot, and C. Adams, "Usability of Anonymous Web Browsing: An Examination of Tor Interfaces and Deployability," in *SOUPS*, Pittsburgh, USA, July 2007.

[22] ISO/IEC, *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality measure elements*, 2012, 25021.

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

[24] S. Harrer, J. Lenhard, and G. Wirtz, "BPEL Conformance in Open Source Engines," in *IEEE SOCA*, Taipei, Taiwan, December 17-19 2012.

[25] ——, "Open Source Versus Proprietary Software in Service-Orientation: The Case of BPEL Engines," in *ICSOC*, Berlin, Germany, December 2013, in press.

[26] D. Spinellis, *Quality Wars: Open Source Versus Proprietary Software*. O'Reilly Media, Inc., 2011, Making Software, ISBN: 978-0-596-80832-7.

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