# Towards Uniform BPEL Engine Management in the Cloud

Simon Harrer, Jörg Lenhard, Guido Wirtz

Distributed Systems Group
University of Bamberg
An der Weberei 5
96047 Bamberg
Germany
firstname.lastname@uni-bamberg.de

Tammo van Lessen

innoQ Deutschland GmbH
Krischerstr. 100
40789 Monheim am Rhein
Germany
tammo.van-lessen@innoq.com

**Abstract:** The *Web Services Business Process Execution language* (BPEL) is a standard for modeling and executing automated processes and is tailor-made for service orchestration. BPEL specifies a serialization format which every BPEL implementation has to understand, thus allowing for the portability of processes among runtime engines. Although the modeling and execution of BPEL processes is portable between engines to a large degree, the lifecycle management of BPEL processes is not standardized and varies a lot for different engines. This paper presents a first approach for a uniform and cloud-based lifecycle management of BPEL processes and engines. We infer a uniform interface for the lifecycle management from the capabilities of current engines and provide a prototypic implementation of a tool that manages processes and engines on a TOSCA-compliant infrastructure.

## 1 Introduction

The *Web Services Business Process Execution Language 2.0* (WS-BPEL, or BPEL for short) [OAS07] is an OASIS standard specifying a process language with which executable Web Services-based orchestrations can be modeled. If these orchestrations or processes are modeled in a standard conformant manner, they can be executed on any standard conformant BPEL engine, ensuring portability and avoiding vendor lock-in. Similar to Java's concept of *write once, run anywhere*, BPEL was developed with the mantra *model once, run anywhere* in mind [KKL06]. However, the standardization initiative did not take all management related tasks into account. In terms of the BPM lifecycle [vdAtHW03] depicted in Fig. 1, the phases of *system configuration*, *process enactment*, and *diagnosis* are not standardized and differ for every BPEL implementation. As a consequence, the seemingly simple task of deploying a BPEL process on a BPEL engine requires engine dependent meta data files, called deployment descriptors, to be present (phase system configuration) and engine specific deployment methods have to be called (phase process enactment). Also tasks such as monitoring and log file formats (phase diagnosis) are not standardized. This aggravates the automation of deploying BPEL processes in a cloud environment, as every user has to automate these tasks for every engine they use. This situation is typical for these types of

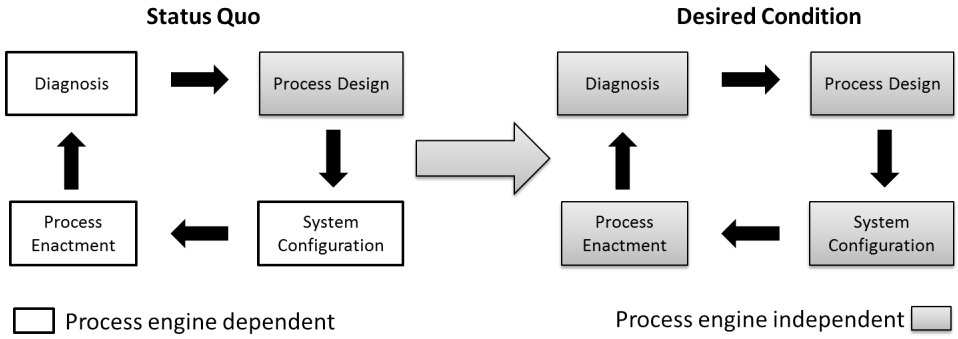


Figure 1: The status quo and the desired condition of the BPM lifecycle (adapted from [vdAtHW03])

systems and no different for the implementations of other process specifications, such as the *Business Process Model and Notation* (BPMN) [OMG11].

Since the finalization of BPEL in 2007, a plethora of open source and proprietary BPEL engines have emerged. These have a varying degree of standard conformance, as demonstrated in recent studies [HLW12, HLW13], which implies a varying degree of portability for BPEL processes in general. In other words, these studies revealed the strengths as well as weaknesses of the available BPEL engines and the need to select an engine according to the BPEL activities used in a BPEL process. For instance, some engines do not support concurrent execution of activities, which impedes the execution of processes that make use of concurrency. Because of this, a company may use multiple engines for different processes depending on their feature requirements, which in turn increases the maintenance effort, as administrators have to manage multiple engine installations at the same time. We aim to overcome these issues by providing a uniform BPEL management layer for these tasks to reduce maintenance costs and to enable the use of BPEL engines in the cloud. It can be expected that this uniform layer can be applied to engines for other process languages as well.

TOSCA [OAS13b] is an emerging OASIS standard for a holistic management of complex application topologies in cloud environments and therefore a good fit for the task at hand. It describes the application's structure using a typed graph: Node Templates represent the components of the application and relationship templates the relationships between the components. Each node template has a node type. A node type may offer interfaces to manage its lifecycle [BBLS12]. The presented BPEL management layer is a first step towards a BPEL engine node type allowing a uniform management of BPEL engines by a TOSCA runtime.

The paper is structured as follows. The next section discusses related work on the automated deployment and management of applications in cloud environments in general and of BPEL processes in particular, as well as on the assessment of BPEL portability. Section 3 presents the main contribution of the paper: A Web service-based approach to manage BPEL engines, the deployed processes and their instances using a uniform interface. Section 4 outlines the

implemented prototype. Finally, Section 5 concludes the paper and provides an outlook on future work.

## 2 Related Work

Related work separates in work on the automated deployment and management of BPEL processes, the assessment of their portability, the deployment of complex application topologies using TOSCA and the usage of uniform APIs in cloud computing.

The BPEL engine test system (*betsy*[1]) [HL12] consists of a test suite to evaluate the conformance of BPEL engines to the BPEL specification, as well as a tool to automatically determine the conformance degrees by running the test suite against multiple BPEL engines. Hence, betsy implements the logic to manage multiple BPEL engines as part of its test bed. This includes the (re)installation, startup, shutdown, the deployment of a process, and the retrieval of the log files of an engine. In [HRW14], Harrer et al. extended betsy with the creation of *vbetsy*, a tool that tests BPEL engines that are being provisioned and run in dedicated virtual machines in an effort to allow for a more efficient and fast testing process. As part of this extension, they extracted two interfaces: the `EngineLifecycle` interface to install, start, and stop the engine, and the `EngineActions` interface to deploy a process and retrieve the log files of the engine. The methods of the `EngineActions` interface are accessible via the network by exchanging Java objects over TCP, while the `EngineLifecycle` methods are only accessible locally. In our approach, we reuse these existing methods and integrate them into a holistic approach for uniform BPEL management. Moreover, instead of accessing these methods locally or via exchanging Java objects over TCP, we provide portable and platform-independent WSDL-based Web services for all engine capabilities. Similar to this, [vLLM+08] describes a management framework for BPEL engines. However, this framework is more of a suggestion for engine vendors how such a management framework should look like and is not aiming at the unification of proprietary management APIs. Here, we do not propose a framework to be implemented by engine vendors, but build a unified framework on top existing proprietary management APIs, without requiring these to be adapted.

In [HLW12, HLW13] betsy is used to evaluate the conformance of five open source and three proprietary engines by means of the test suite described in the previous paragraph. These results have been used in [LW13] to compute the degree of portability of a BPEL process using software metrics and to evaluate the quality of four mappings from several modeling languages to BPEL [LW13]. This computation is done automatically by the *bpp*[2] tool. This tool is a static analyzer that parses BPEL processes and detects portability issues in them. As part of this work, we extended and integrated the tool to support the automatic selection of a suitable BPEL engine for a specific BPEL process by rejecting the engines that do not support the BPEL features used in the process. A similar, policy-based selection approach is described in [MvLW+09]. However, it is focused on the automatic routing

[1] The tool is open source and publicly available at `https://github.com/uniba-dsg/betsy`.

[2] Bpp stands for *BPEL Portability Profile*. The tool is open source and publicly available at `https://github.com/uniba-dsg/bpp`.

of requests to the best suited service implementation and not on finding a suitable BPEL engine for a certain process.

Lego4TOSCA [HLNW14] presents a generic architecture for the implementation of TOSCA node types. Concrete node type implementations are called *implementation artifacts*, and are designed to work together to realize management functionality. For instance, the Apache Tomcat implementation artifact uses the Windows and the Ubuntu implementation artifact to execute operations on the respective virtual machine.

In cloud computing, the management interfaces of cloud vendors vary greatly despite the fact that they support a common set of management operations [KW14]. Hence, there is a trend towards uniform APIs for both the management of Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) environments to provide a uniform access to these common operations. For instance, the *Open Cloud Computing Interface* (OCCI) [OCC11] provides a uniform API for IaaS, whereas *Cloud Application Management for Platforms* (CAMP) [OAS13a] proposes a uniform API for PaaS. Our approach solves a similar problem regarding BPEL engine vendors and their APIs, i.e., the provisioning, deployment and monitoring of infrastructure and services. In addition, a uniform API for BPEL engines in the form in which we propose it, can easily interface with other uniform APIs and be integrated in cloud provisioning and deployment plans, e.g., using TOSCA.

## 3 Approach

In this section, we present our idea of a uniform BPEL management layer, called *UBML*, for accessing all BPEL-related services. The big picture is depicted in Figure 2 which outlines the relationship of the management layer to multiple BPEL engines and a typical usage scenario in the form of a standard conformant BPEL process that should be executed on a suitable engine. The UBML comprises seven services which abstract the common callable functionality of the BPEL engines and provides an API to the user which supports multiple application scenarios. One of these scenarios, which can be seen as *typical* usage scenario is shown by the numbered edges from the standard conformant BPEL process to the different UBML services in Fig. 2. In the first step, we select the best available engine for this particular process using the *Engine Selection* service. Next, we install and start the selected engine via the *Engine Provisioning* and *Engine Lifecycle* components, respectively. The *Process Deployment* component can then be used to deploy the standard conformant process onto this running engine. This component takes care of every aspect of the deployment, including the generation of a valid and engine dependent deployment descriptor. When the process is now deployed, we can monitor its runtime execution via the log files (*Logfile Access*) of the engine, perform the recovery of activities (*Process Management*) and access the *Audit Trail*.

As the uniform BPEL management layer abstracts from all engine specific APIs, we are able to provide the services of this layer as Web services which can be integrated into Cloud-based provisioning processes. Moreover, we contribute to the effort of maintaining the portability advantages of the BPEL specification: *model once, run anywhere*.

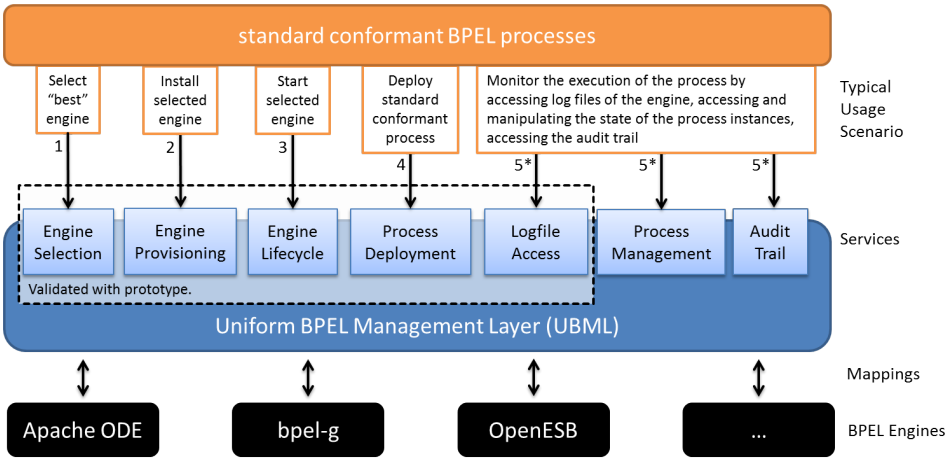


Figure 2: The Uniform BPEL Management Layer (UBML)

This section is organized according to the order in which the services are used in the typical usage scenario. At first, the *Uniform Engine Selection* service is detailed in Section 3.1 followed by the *Uniform Engine Provisioning* and the *Uniform Engine Lifecycle* in Section 3.2 and Section 3.3, respectively. Next, Section 3.4 describes the *Uniform Process Deployment* whereas the three remaining sections (3.5, 3.6 and 3.7) outline our approach on uniform monitoring, activity recovery, and troubleshooting.

### 3.1 Uniform Engine Selection

BPEL engines differ in terms of their standard conformance (e.g., the support for the various BPEL language elements, such as the `<validate>` activity), their functional capabilities (e.g., some engines are able to invoke REST services via proprietary extensions or provide special support for in-memory enactment to improve the performance of process execution) or their nonfunctional requirements (e.g., support for additional Web Services standards, such as WS-ReliableMessaging [OAS09] and WS-Security [OAS06]). To automatically select the engine which is providing the best service for the given BPEL process, we take two kinds of data into account: First, we analyze the language elements the process uses and match it against a standard conformance database, which contains information about which language element is supported by which engine. The language benchmarks of [HLW12, HLW13] are the primary source for this database and a static analyzer tool proposed and used in [LW13, LW13] performs the analysis of the process. This way, we can determine which engines are certain to be unable of supporting the process to be deployed and can restrain from using these engines in the following steps. Moreover, we can rank engines according to the amount of standard conformance they provide. Second, we use policy attachments to describe functional or nonfunctional requirements in an abstract manner. That way, for each engine, its requirements and capabilities are captured using

a policy description language, such as WS-Policy. Furthermore the process models to be deployed can be enriched with policies describing their requirements on a suitable engine. Such policies may define alternative sets of requirements of which at least one must be supported and may be embedded directly into the process via extensions or be attached to it as a separate file.

Based on the policy descriptions, the deployment component will find and select the best matching engine (applying a configurable ranking algorithm in case multiple engines match the requirement) and will compute an effective policy (i.e., the set of common requirements and capabilities). This effective policy can then be used to generate the vendor specific deployment descriptor to configure a certain behavior.

### 3.2 Uniform Engine Provisioning

After a suitable BPEL engine has been selected the engine needs to be provisioned on an appropriate hardware and operating system. Some engines have preferences on which hardware or operating system they run in a more optimized manner, which has to be taken into account for the engine provisioning process. Moreover, at installation time, configuration options must be possible to set, e.g., the port on which the engine is accessible. Furthermore, as the engines may require additional software, e.g., databases, container, etc., these have to be taken into account as well. In summary, the *Engine Provisioning* service takes care of selecting the appropriate server and operating system and is able to install or uninstall the engine and its dependent components. To achieve this, it needs to know which steps are needed for a successful (un)installation for each engine and will expose this functionality through a generic interface.

### 3.3 Uniform Engine Lifecycle

When BPEL engines are provisioned automatically, it is important to also manage their lifecycle in a uniform fashion. The *Uniform Engine Lifecycle* service exposes a generic interface, which allows for starting and stopping all known engines. That way, the UBML can automatically manage the resource consumptions of the managed engines and smartly decide whether an engine can be stopped (i.e., only if there are no processes deployed anymore).

### 3.4 Uniform Process Deployment

The deployment of a BPEL process to an engine is nontrivial, as it requires the creation of an engine specific deployment descriptor and the usage of an engine specific deployment method. As shown in [HL12, LHW13], there are major differences in the complexity of the deployment descriptors. There are engines which do not require additional descriptors,

but also others that require three per process. Regarding the deployment methods, there are multiple strategies as well, for instance deployment through a command line interface, through a Web service or manually through the browser. We aim to overcome these differences by a) generating deployment descriptors automatically based on the information already present in the BPEL and WSDL files, and b) by providing a uniform Web service for deploying processes on a total of seven engines in different configurations and versions.

### 3.5 Uniform Logfile Access

The location of the log files depends heavily on the engine and environment. Although most engines have a single log file, there typically also are multiple logs for the environment or container in which the engine is running. As all these files may be scattered into different places for each engine, there is a need to retrieve the files for troubleshooting and maintenance in a uniform way. We capture these files using an engine independent Web service that collects all files and returns them as a collection. That way, the IT support staff has a single point of access for all troubleshooting information.

### 3.6 Uniform Process Management

Most BPEL engines provide sophisticated management APIs. These interfaces enable applications to suspend/resume, debug or recover process instances. In addition, some of these engines allow for manipulating instance data (i.e., variables and partner links) or to rewind a process to a specific point in the past and to resume it from this position. Our management component aims at unifying these APIs in order to provide a single API that is capable to monitor all supported engines.

### 3.7 Uniform Audit Trail

Audit trails are an important asset when analyzing the execution of processes for a purpose such as process mining. Several tools such as ProM[3] or fluxicon's Disco[4] support that task but in some cases the only requirement is to display which parts of the process have been properly executed already. Most engines are able to provide a detailed audit trail but unfortunately there is no commonly implemented standard for their format. When abstracting from the concrete engine that is actually executing the given process, there is also the need to unify the audit trail. In order to achieve this, our monitoring component can translate engine specific audit data to a generic audit trail data format such as MXML [Don05].

[3] For more information, visit the project page at `http://www.promtools.org/prom6/`

[4] For more information, visit the project page at `http://www.fluxicon.com/disco/`

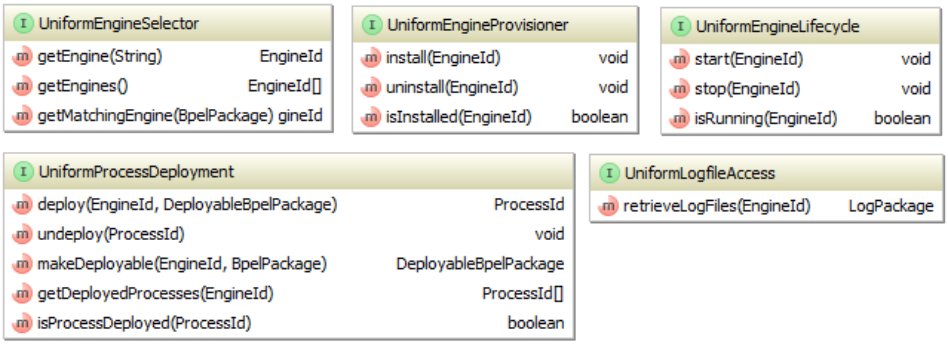


Figure 3: Class Diagram of the Uniform BPEL Management Layer implemented by the prototype.

## 4 Prototype

The prototype discussed in this section validates the feasibility of our proposed approach from Section 3. As shown in Figure 2, our tool implements the five UBML services: *Engine Selection*, *Engine Provisioning*, *Engine Lifecycle*, *Process Deployment* and *Logfile Access*. For each of these services, the prototype provides a callable Web service that maps the uniformly defined actions to engine specific calls. At the moment, UBML supports seven different open source BPEL engines in different configurations (e.g., in-memory or persistent execution) and versions. In particular, we support Apache ODE in versions 1.3.5 and 1.3.6 with the in-memory option, bpel-g in version 5.3 with the in-memory option, ActiveBPEL v5.0.2, Orchestra v4.9, OpenESB in versions 2.2, 2.3 and 2.3.1, Petals ESB in versions 4.0 and 4.1 and WSO2 Business Process Server versions 2.1.2, 3.0.0, and 3.1.0. The interfaces of the five services can be seen in the class diagram in Figure 3. The services are published as WSDL 1.1 Web services, and group low-level methods according to their task. Engines are identified with the string-based `EngineId` while deployed BPEL processes are referred to using the QName[5]-based `ProcessId` which also contains an `EngineId`. The standard conformant BPEL process and its related files (e.g. WSDL, XSD or other files) are sent to the interfaces as a `BpelPackage` without any engine specific information while the `DeployableBpelPackage` is an archive that includes the relevant files and structure to be deployable on a specific engine. Log files are transferred similarly in the form of packages using the `LogPackage` class.

The *Engine Selector* service is implemented as a registry or repository. It either provides a list of all available engines or the best engine for a specific BPEL process. For determining the latter, we use an extended version of the *bpp* tool to which we transfer the path to the BPEL file within the `BpelPackage`. Using the conformance data of the engines, the tool returns a list of engines that are able to support the process. Moreover, it also provides the number of supported features per engine, thus, providing the data to create a ranking of the engines that support the process by their overall support of the specification from which the topmost is selected. The other four services are implemented by reusing and extend-

[5]QName stands for qualified name, i.e., an identifier with a namespace [W3C09].

ing the engine specific logic of *betsy* [HL12]. The `UniformEngineProvisioning` component allows installing and uninstalling any supported engine as well as checking whether it is installed or not. To start, stop, or check the status of an engine, our tool provides corresponding methods in the `UniformEngineLifecycle` service. The largest service is the `UniformProcessDeployment` service, which supports the deployment and undeployment of a standard conformant BPEL process. The deployment, however, requires a preprocessing step as the `BpelPackage` has to be made deployable via the `makeDeployable` method which returns the desired `DeployableBpelPackage` that is required by the `deploy` method. Whether a process is already deployed can be checked with the method `isProcessDeployed` while a list of all deployed processes of a single engine can be retrieved via `getDeployedProcesses`. The smallest component is the `UniformLogfileAccess` which simply returns a package with all the log files for a specific engine.

Listing 1: Composition of UBML Services

```
1  class CompositeProcessProvisioningService  {
2      ProcessId makeProcessAvailable(BpelPackage bpelPackage) {
3          EngineId engineId = UniformEngineSelection.getMachingEngine(
               bpelPackage);
4          if(!UniformEngineProvisioning.isInstalled(engineId)) {
5              UniformEngineProvisioning.install(engineId);
6          }
7          if(!UniformEngineLifecycle.isRunning(engineId)) {
8              UniformEngineLifecycle.start(engineId);
9          }
10         DeployableBpelPackage deployableBpelPackage =
               UniformProcessDeployment.makeDeployable(engineId,
               bpelPackage)
11         return UniformProcessDeployment.deploy(engineId,
               deployableBpelPackage );
12     }
13     void makeProcessUnavailable(ProcessId processId) {
14         UniformProcessDeployment.undeploy(processId);
15         EngineId engineId = processId.getEngineId();
16         ProcessId[] processIds = UniformProcessDeployment.
               getDeployedProcesses(engineId);
17         if(processIds.length == 0) {
18             UniformEngineLifecycle.stop(engineId);
19             UniformEngineProvisioning.uninstall(engineId);
20         }
21     }
22 }
```

Based the lower-level operations depicted in Figure 3, we can build the functionality outlined in the typical usage scenario from Figure 2 and shown in Java-based pseudo code in Listing 1 as part of the `CompositeProcessProvisioningService` class. In the method `makeProcessAvailable`, we only need to pass in a `BpelPackage` and get the `ProcessId` of the, then running, process back. This operation orchestrates the *Engine Selector* component to select an engine that best supports the BPEL features of the provided process, the *Engine Provisioning* service to install the selected engine if it is not already installed, the *Engine Lifecycle* service to start the selected engine if it is not already started, and the *Process Deployment* component to create the `DeployableBpelPackage` and


Figure 4: The EngineControl GUI for controlling engine related tasks.

deploy it. In contrast, the method `makeProcessUnavailable` implements the reverse logic which allows to undeploy a process using the *Process Deployment* service and freeing resources. If the engine which previously undeployed the process has no other processes deployed, we can easily terminate it via the *Engine Lifecycle* service (i.e., freeing RAM and CPU power) and uninstall it via the *Engine Provisioning* service (i.e., freeing disk space). Furthermore, multiple other additional services and operations can be built, e.g., a dashboard showing the list of all currently installed and started engines and the deployed and running processes.

Our prototype is open source, publicly available[6] and written in Java 8 and Groovy 2.3. The Web services are exposed using the JAX-WS[7] 2.0 Java API and can be started using the `ubml.ws.WSMain` class, which sets up the Web services using the following pattern `http://localhost:1234/INTERFACE_NAME`. These addresses can be integrated into TOSCA plans to enable the uniform management of BPEL-related tasks. Using the Gradle[8] task `run`, the Web services start automatically. By executing `gradlew run`, Java 8 is the only runtime dependency, everything else is installed automatically on demand. This enables an easy deployment of this tool on any virtual machine to set up BPEL engines in the cloud.

Furthermore, the tool also contains a simple GUI application which can be started via the `gradlew enginecontrol` command on the command line. This GUI can control the operations of the three services *UniformEngineProvisioning*, *UniformEngineLifecycle*, and *UniformLogfileAccess* as shown in Figure 4.

Several limitations still remain for this prototype. First, only a single engine can currently be run on the same machine, because the engines run on their default ports, which can create conflicts. In the future, we aim to provide configuration options during provisioning to fix this. Second, the automatic creation of the deployable BPEL package including the deployment descriptors does only work for BPEL processes from the betsy test suite. We want to extend this feature to arbitrary BPEL processes in the future, however, this is

[6] See the project page for more information: `https://github.com/uniba-dsg/ubml`

[7] JAX-WS 2.0 is described in the Java Specification Request 224 which is available at `https://jcp.org/en/jsr/detail?id=224`.

[8] Gradle is a build management tool with which the UBML prototype is built. It is available at `http://www.gradle.org/`.

considered quite time consuming as this requires parsing much information from the BPEL and WSDL files. Third, both the `undeploy` and the `isDeployed` commands are only implemented for Apache ODE. We aim to implement the mapping for the other remaining six open source engines as well. Fourth, the prototype only runs on Windows 7 64bit. As many virtual machines use a Linux-based operating system, we plan to port the mappings to run on Linux as well.

## 5 Conclusion and Future Work

This paper presents a Web Services-based approach to manage BPEL engines and deployed processes and we provide a prototypic implementation of the approach. The current limitation is that the logic to manage the BPEL engines has to be installed manually on the virtual machine where the engine should be deployed on. Our plan is to leverage virtualization, i.e., port the prototype to the Lego4TOSCA universe. This allows us to provision arbitrary virtual machines without any user intervention. Moreover, as this approach only supports open source BPEL engines, we aim to support proprietary BPEL engines as well. Another subject of future work is the construction of a unified monitoring API across all vendors. Finally, the presented work focuses on BPEL engines and our next step is to apply the presented concepts on process engines for BPMN [OMG11].

**Acknowledgment** We would like to express our gratitude to Oliver Kopp for fruitful discussions on this topic.

## References

[BBLS12] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(03):80–85, May 2012.

[Don05] B. F. Van Dongen. A Meta Model for Process Mining Data. In *In Proceedings of the CAiSE WORKSHOPS*, pages 309–320, Porto, Portugal, 2005.

[HL12] Simon Harrer and Jrg Lenhard. Betsy–A BPEL Engine Test System. Technical Report 90, Otto-Friedrich Universitt Bamberg, July 2012.

[HLNW14] Florian Haupt, Frank Leymann, Alexander Nowak, and Sebastian Wagner. Lego4TOSCA: Composable Building Blocks for Cloud Applications. In *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD 2014)*, Anchorage, Alaska, USA, 2014. IEEE.

[HLW12] Simon Harrer, Jrg Lenhard, and Guido Wirtz. BPEL Conformance in Open Source Engines. In *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'12), Taipei, Taiwan*, pages 1–8. IEEE, 17–19 December 2012.

[HLW13] Simon Harrer, Jrg Lenhard, and Guido Wirtz. Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines. In *ICSOC*, volume 8274 of *Lecture*

*Notes in Computer Science*, pages 99–113, Berlin, Germany, 2013. Springer Berlin Heidelberg.

[HRW14] Simon Harrer, Cedric Rck, and Guido Wirtz. Automated and Isolated Tests for Complex Middleware Products: The Case of BPEL Engines. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 390 – 398, Cleveland, Ohio, USA, April 2014. Testing Tools Track.

[KKL06] R. Khalaf, A. Keller, and F. Leymann. Business processes for Web Services: Principles and applications. *IBM Systems Journal*, 45(2):425–446, 2006.

[KW14] Stefan Kolb and Guido Wirtz. Towards Application Portability in Platform as a Service. In *Proceedings of the 8th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, Oxford, United Kingdom, April 7–10 2014. IEEE.

[LHW13] Jrg Lenhard, Simon Harrer, and Guido Wirtz. Measuring the Installability of Service Orchestrations Using the SQuaRE Method. In *Proceedings of the 6th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'13)*, Kauai, Hawaii, USA, December 16-18 2013. IEEE.

[LW13] Jörg Lenhard and Guido Wirtz. Measuring the Portability of Executable Service-Oriented Processes. In *Proceedings of the 17th IEEE International EDOC Conference*, pages 117 – 126, Vancouver, Canada, September 2013. IEEE.

[MvLW+09] Ralph Mietzner, Tammo van Lessen, Alexander Wiese, Matthias Wieland, Dimka Karastoyanova, and Frank Leymann. Virtualizing Services and Resources with ProBus: The WS-Policy-Aware Service and Resource Bus. In *Proceedings of the 7th International Conference on Web Services (ICWS) 2009*, Los Angeles, CA, USA, July 2009. IEEE Computer Society.

[OAS06] OASIS. *Web Services Security*, February 2006. v1.1.

[OAS07] OASIS. *Web Services Business Process Execution Language*, April 2007. v2.0.

[OAS09] OASIS. *Web Services Reliable Messaging*, February 2009. v1.2.

[OAS13a] OASIS. *Cloud Application Management for Platforms*, July 2013. Version 1.1 – Draft 03.

[OAS13b] OASIS. *Topology and Orchestration Specification for Cloud Applications*, November 2013. v1.0.

[OCC11] OCCI. *Open Cloud Computing Interface - Core*. Open Grid Forum, 2011.

[OMG11] OMG. *Business Process Model and Notation*, January 2011. v2.0.

[vdAtHW03] Wil M. P. van der Aalst, Arthur H.M. ter Hofstede, and Mathias Weske. Business Process Management: A Survey. In *Proceedings of the International Conference on Business Process Management*, Eindhoven, The Netherlands, 2003. Springer Berlin Heidelberg.

[vLLM+08] Tammo van Lessen, Frank Leymann, Ralph Mietzner, Jörg Nitzsche, and Daniel Schleicher. A Management Framework for WS-BPEL. In *Proceedings of the 6th IEEE European Conference on Web Services 2008*, pages 187–196, Dublin, Ireland, November 2008. IEEE Computer Society.

[W3C09] W3C. *Namespaces in XML*, December 2009. v1.0.