Automated and Isolated Tests for Complex Middleware Products: The Case of BPEL Engines

Simon Harrer, Cedric Röck and Guido Wirtz
Distributed Systems Group
University of Bamberg
Bamberg, Germany
{*simon.harrer,guido.wirtz*}@*uni-bamberg.de*
cedric-marcel.roeck@stud.uni-bamberg.de

***Abstract*—Today, a plethora of enterprise middleware solutions are available, leading to the problem of choosing the right tool for a specific use case. Automated tests can support the selection of such software by determining decision relevant metrics, like e.g., throughput or the degree of standard conformance. To avoid side effects between tests, test isolation, i.e., to provide fresh instances of the software for each test execution, is essential. However, middleware suites are inherently complex, provide a large range of configuration options, have tedious or sometimes manual installation procedures, and long startup times. These idiosyncrasies aggravate the creation of fresh instances of such middleware suites, leading to slower turnaround times and increasing the cost for ensuring test isolation. We aim to overcome these issues with methods and tools from the area of virtualization and devops. In this work, we focus on BPEL engines which are common middleware components in Web Service based SOAs. We applied our proposed method to the BPEL Engine Test System (*betsy*), a conformance test suite and testing tool for BPEL engines. Results reveal that our method a) enables automatic creation of fresh instances of software without manual installation steps, b) reduces the time to create these fresh instance dramatically, and c) introduces only a neglectable performance overhead, therefore, reducing the overall costs of testing complex software.**

***Keywords*-test isolation, test automation, virtualization, BPEL engines**

I. Introduction

Today, middleware suites provide the foundation of almost any enterprise IT landscape. In 2012, Gartner estimates the market for such middleware products at \$20 billion[1]. In this market, a plethora of middleware suites exist which differentiate in both functional and non-functional capabilities. This variety ensures that the selection of a suitable middleware is itself a "nontrivial project" [1, p. 88]. Instead of manually checking the suitability of multiple middleware suites, we can leverage automated tests to automatically calculate decision relevant metrics [1]. For example, the degree of standard conformance helps determining the functional capabilities whereas the performance metrics, e.g., transactions per second or execution time, helps determining the non-functional capabilities. To guarantee correct results of test suites producing such metrics, we need to avoid any side effects between test cases, that is, we need to ensure test isolation [2]. The safest way to guarantee full test isolation is to provide a fresh instance by reinstalling and starting the software for each test case. This prevents any possible side effects from one test to another. However, due to the purpose and role of middleware systems, these software components are inherently complex [3]. This manifests itself in multiple forms, e.g., the high number of configuration parameters, a tedious as well as time consuming installation processes, and long startup times. These idiosyncrasies aggravate benchmarking of such software systems in two ways. First, a tedious and complex installation processes may include manual steps which may not be that easily automated. This is problematic as the reliability and repeatability of tests depends on their ability to be automated. Second, the time until the next fresh instance is available for testing is the sum of the installation and the startup time. This test preparation time is engine dependent and can vary greatly. If this time is high, the test turnaround time per test is high as well. We present and answer the following research question in our paper:
How to provide a fresh and started instance of a complex software in a time efficient and effective manner automatically?

As the field of different middleware systems is wide, we focus on one particular type of component in modern middleware suites, namely process engines in general and BPEL engines in particular. The BPEL 2.0 specification [4] is the de facto standard for orchestrating services within a Web Service based Service-oriented Architecture (SOA) [5]. The language defined in the specification provides control-flow and data-flow activities to implement such service orchestrations by means of exchanging SOAP messages. Since the finalization of the standard in 2007, multiple open source and commercial BPEL engines have emerged, leaving the developer with an agony of choice which can be mitigated by automated tests [6]. These engines, however, cannot be

[1]Market Share: All Software Markets, Worldwide, 2012, Colleen Graham, et al, March 29, 2013

executed standalone, but have to be either deployed to a managed environment with diverse complexity, being it a servlet container, an application server or an enterprise service bus (ESB) [7]. Consequently, these BPEL engines inherit the issues of their managed environment, i.e., having tedious and time intensive installation processes and long startup times. In open source BPEL engines, the installation time ranges from approx. three seconds up to more than two minutes [8]. The installation of commercial products in this field is even more tedious. The Oracle Business Process Engine, for example, is part of the Oracle SOA Suite 11gR1 middleware. The guide that describes the installation[2] requires the user to a) download five files summing up to five GB in total[3] and b) follow the necessary installation steps described on 48 pages. The issue of the startup time is not that pressing as the issue of installation, but should not be neglected. While the open source engines start in at most one minute, commercial ones within large middleware suites may have much longer startup times. The Oracle middleware suite requires more than ten minutes until it is up and running[4].

The BPEL Engine Test System (*betsy*) can automatically evaluate the conformance and the expressiveness of such BPEL engines using test cases [6]. Once, one of its test cases caused an infinite loop within one BPEL engine which in turn lead to the failure of all subsequent test cases within the test suite [7]. As a consequence, *betsy* adopted a straight-forward approach to achieve test isolation: it creates a fresh and started instance of a BPEL engine by removing the previously installed engine and reinstalling as well as starting it. This approach, however, is not scalable. Harrer et al. [9] evaluated five open source BPEL engines regarding standard conformance with approximately 130 test cases. This evaluation took 10 hours to complete. Since the publication, the number of tests (additional test suites) and engines (different configurations, commercial ones) has increased [6]. This caused an increase in time for testing a single engine as well as all engines together.

In this work, we aim to find ways to reduce the increasing test time to get feedback more quickly. In particular, we want to reduce the most time intensive tasks, namely, the install and startup time of the BPEL engines. With *vbetsy*, our extension of the testing tool *betsy*, we are able to reduce overall test execution time, including the three test phases *setup*, *test execution* and *teardown*, by up to 93% while still retaining test automation and isolation. This is done by reducing the installation time of each BPEL engine by encapsulating each engine within a reusable VM image and the startup time of each BPEL engine by creating and then restoring a snapshot of a VM with an already started engine. This approach guarantees test isolation, as for every test a freshly restored instance of a VM is provided. Moreover, it also guarantees test automation, as we provide a) scripts to automatically create and provision new VMs and b) interfaces to import, start and stop VMs as well as create and restore snapshots from within *betsy*.

The paper itself is structured as follows. In section II, we discuss related work, namely, middleware and BPEL engine benchmarking, automated provisioning of VMs and test isolation using VMs. Next, we give a short description of *betsy* and its current approach on creating fresh engine instances in section III. In section IV, we provide details on *vbetsy*, the extension of *betsy* with virtualization techniques and devops[5] methods, followed by an evaluation of the performance improvements in section V. The paper closes with a discussion of the limitations of our approach in section VI and a summary and future work in section VII.

II. Related Work

Related work is subdivided into (A) benchmarking middleware and BPEL engines, (B) provisioning of VMs and (C) the usage of VMs for testing purposes.

A. Benchmarking Middleware and BPEL Engines

Benchmarking middleware and BPEL engines is a wide field, therefore, we focus solely on performance and conformance evaluations.

Testing middleware components, e.g., ESBs, Java Messaging Systems (JMS) and BPEL engines, under heavy load is an area of some interest [10], [11], [12], [13], [14], [15]. Regarding BPEL engines, three different performance studies are available. Bianculli et al. [13] present *SOABench* which allows to generate testbeds for testing SOA applications. Based on their testbed generation tool, they compared the performance of four BPEL processes on three BPEL engines in [14]. In contrast, Roller [15] focuses on benchmarking solely a single commercial BPEL engine with the aim of improving its performance based on his previously obtained test results. As part of the development of a BPEL engine for mobile devices called *sliver* [16], Hackmann et al. conducted a performance comparison of sliver and ActiveBPEL based on twelve test cases. However, these three approaches do not use any automated test isolation strategy.

Regarding conformance testing, much work has been done in the area of Java Enterprise Edition (JEE). Oracle provides Test Compatibility Kits (TCKs) which can be used to evaluate whether a servlet container implements the servlet specification, however, provides no test isolation mechanisms. Lately, the conformance of BPEL engines has been of interest. In [6] and [9], open source as well as commercial BPEL engines were put under scrutiny by evaluating which of the BPEL features and of the workflow patterns [17] from van

[2]The installation guide is available at http://bit.ly/soasuitequickstartguide

[3]See page five and six at http://bit.ly/soasuitequickstartguide

[4]This was measured by starting the AdminServer in the prebuilt virtual machine (VM) provided from Oracle at http://www.oracle.com/technetwork/middleware/soasuite/learnmore/vmsoa-172279.html

[5]For a definition see Gartner, Hype Cycle for Cloud Computing, 2011

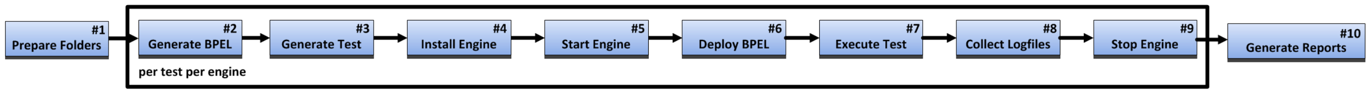

Figure 1. *betsy* Test Procedure. Adopted from [9].

der Aalst et al. they support. This work is based upon the tool used in these publications.

B. Provisioning of Virtual Machines and Appliances

In the area of provisioning applications within VMs, there are basically two approaches available. The most popular approach is based on script-based solutions, e.g., *Puppet*[6], *chef*[7], and *sprinkle*[8]. These have in common the steps required to install and verify the correctness of the installation of a specific component are specified in a module. These modules may then be reused and reconfigured for more complex environmental infrastructure. Lately, however, a more service-based solution has emerged based upon the OASIS standard Topology and Orchestration Specification for Cloud Applications (TOSCA) [18]. Instead of implementing these steps in scripts, the steps are implemented as BPEL processes. This has the advantage that visualizing the provisioning process as well as monitoring its execution is basically free. For TOSCA, there exists a reference implementation of the modeling tool [19] and the runtime [20]. For *vbetsy*, we leverage *sprinkle* for the provisioning scripts while aiming to convert them to TOSCA processes in the future.

C. Testing with Virtual Machines (VMs)

Since the emergence of virtualization technology, many approaches have adopted the use of VMs for enabling or speeding up testing. Especially in the area of testing the design of a website in different browsers, it is widely applied[9]. In this case, the system under test is the web application and the VMs are different test cases while we use VMs the other way round. But also in the area of testing middleware, VMs are used. For evaluating the performance of ESBs, the corresponding ESBs are installed on Amazon EC2[10] where the test itself is executed as well. While they use VMs, they do not control them during test execution but just leverage the cheap availability of infrastructure in the cloud instead of buying physical hardware. Another approach which controls VMs and uses snapshot restoration during test is the ruby project Virtual Machine Test Harness[11] (VMTH). Its purpose is the automatic unit testing of provisioning scripts for kernel-based virtual machines (KVMs) running solely on linux. While it also uses VMs and snapshot restoration in a similar way as *vbetsy*, its software under test are provisioning scripts which alter the VM itself while *vbetsy* tests BPEL engines. Due to its focus on its specific domain, VMTH is not reusable for our problem.

III. Status Quo

This work relies on the BPEL conformance test suite and tool *betsy*[12] and extends it. Betsy follows the ten-step test procedure given in Figure 1 and is able to execute this automatically. Step #1 and #10 prepare the output folder structure containing all generated files and generate reports detailing the results of a single test run, respectively. Both steps are executed once for a whole test run while the steps #2 to #9 are executed for every single test case on every engine. In step #2, an engine dependent BPEL process is generated from an engine independent one and converted to a deployable archive. Next, the corresponding *soapUI* test case is generated. In step #4 and #5, *betsy* installs and starts the engine under test locally on the same host as the testing tool is running, followed by the deployment of the previously created archive and the execution of the *soapUI*[13] test case. After executing the test in step #7, the log files of the engine under test are collected for analysis in step #8. The test procedure for a single test case is finished by stopping the previously started engine in the last step.

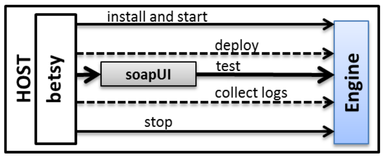

Figure 2. Architecture and Execution Flow of a *betsy* Test Case Execution.

In this context, we focus on the six steps interacting with the engine under test: installing, starting and stopping the engine as well as deploying the BPEL archive, executing the test case and collecting the log files. In Figure 2, these steps are presented as a simplified flow chart highlighting the involved components. In case of executing the test case, *betsy* leverages *soapUI*, while in the five remaining tasks, it interacts directly with the engine under test. These five tasks are grouped in three life cyle tasks and two engine actions as part of *betsy*'s architecture. For this purpose, *betsy* provides the two Java interfaces `EngineLifecycle` and `EngineAction` in Listing 1. These methods are implemented for every local engine using Apache Ant tasks

[6]Available at http://puppetlabs.com/.
[7]Available at http://www.getchef.com/.
[8]Available at http://rubygems.org/gems/sprinkle.
[9]For example at http://browsershots.org/.
[10]See http://esbperformance.org/display/comparison/ESB+Performance
[11]Greg Retkowski. VMTH - Virtual Machine Test Harness. January 2014. url: http://github.com/gregretkowski/vmth.
[12]For a more detailed technical report of *betsy*, see [7].
[13]The tool *soapUI* allows to test functional properties of Web Services.

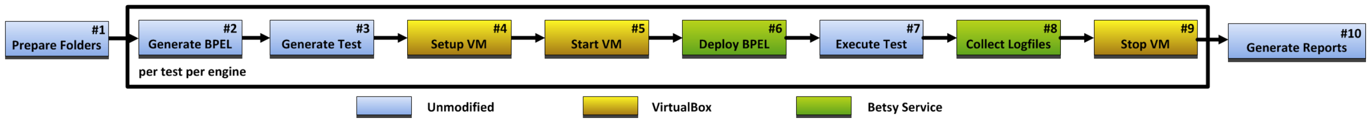

Figure 3. Architecture and Execution Flow of a *vbetsy* Test Case Execution.

and interact with the engines either via CLI, file system operations or Web Service calls.

Listing 1. Common Interface for every Engine
```
interface EngineLifecycle {
    void install(); // incl. re-install
    void startup();
    void shutdown();
    boolean isRunning(); // observable
}
interface EngineActions {
    boolean deploy(BPELArchive archive)
    LogFiles getLogs();
}
```

IV. VM Testing Extension

In this section, we present our main contribution, an extension of the tool *betsy* we call *vbetsy*(*betsy* with virtualization support). The idea is to use VMs and their ability to create and restore previously created snapshots to provide fresh and started instances for each test case execution. *vbetsy* is open source and publicly available[14]. First, we present our life cycle model of a VM in section IV-A. Next, the necessary changes to the test procedure and the architecture of betsy on the basis of the life cycle model of the VMs are shown in section IV-B while the approach on creating the VMs using methods from the devops movement is presented in section IV-C. Last, two issues we faced during development of *vbetsy* are detailed.

A. Our Execution Model of Virtual Machines

To harvest the benefits of using VMs in contrast to the local approach, we need to look at our life cycle model of a VM within *vbetsy* as shown in Figure 4.

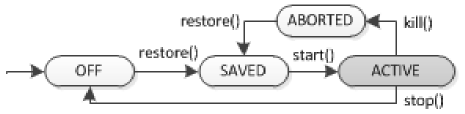

Figure 4. State Chart of our Virtual Machine Lifecycle

From the initial state `OFF`, we can only restore a previously created snapshot. In this snapshot, any software that is required for testing may already be started. From this `SAVED` state, the VM is started. Within this `ACTIVE` state, it is usable exactly for one test case execution. When the test is done, the VM must be stopped returning to its initial state. In case of any unrecoverable errors within the VM, the started instance can also be killed. In both cases, the next step is to restore the snapshot and proceed to the `SAVED` state. Any VM used in *vbetsy* must only change according to our model provided in Figure 4. To guarantee this, *vbetsy* ships with an interface (see Listing 2) that corresponds to the state transitions of the described figure. This interface is implemented by *vbetsy* on top of VirtualBox to control any VM of VirtualBox according to our execution model of VMs.

[14]See https://github.com/uniba-dsg/betsy/tree/icst2014.

Listing 2. Common Interface for our Virtual Machine Execution Model
```
interface VirtualMachineLifecycle {
    void start();
    void stop();
    boolean isActive();
    void kill();
    void restore();
}
```

B. Changes in the Test Procedure and Architecture

With the state machine of the VM life cycle in mind, *vbetsy* modifies the test procedure and architecture. Figure 3, details the changes of the test procedure. The five steps marked blue (#1, #2, #3, #7, and #10) are unchanged while the yellow steps (#4, #5, and #9) are fulfilled by VirtualBox and the green tasks (#6 and #8) by *vbetsy* in conjunction with the *Betsy Service* on the engine VMs. The modification within these five altered steps is best presented in the new flow chart of executing a test case with *vbetsy* as given in Figure 5.

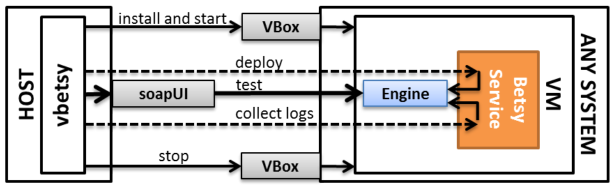

Figure 5. The architecture and execution flow of a *vbetsy* test case execution

As the chart shows, *vbetsy* installs, starts and stops the VM (and the BPEL engine transitively) via the API of VirtualBox, and deploys the BPEL archive to and collects the log files from the engine under test via an API of the *Betsy Service*. This is stark contrast to the execution flow of *betsy* in Figure 2, where *betsy* interacts directly with the engine. As these five tasks are specified in the two interfaces in Listing 1 within *betsy*, *vbetsy* simply provides additional implementations for both interfaces. The methods `start` and `stop` of the `EngineLifecycle` interface are mapped to the equivalent methods of

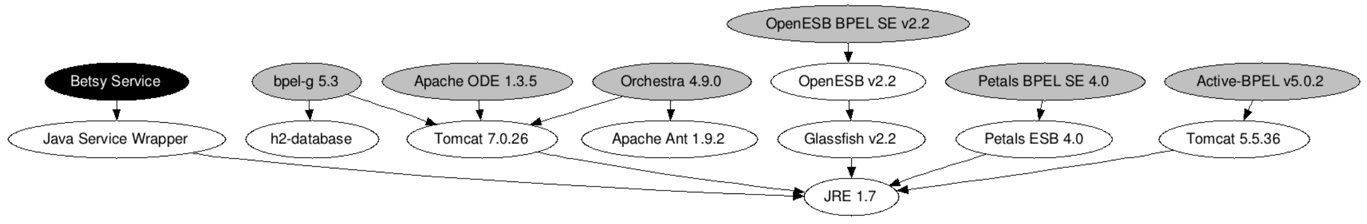

Figure 6. Deployment Topology of all BPEL Engines and the *Betsy Service*

the `VirtualMachineLifecycle` while the `install` method is implemented by means of the `restore` method while in turn, the `VirtualMachineLifecycle` is implemented on top of the VirtualBox Java Web Service. In contrast, the methods of the `EngineActions` interface are implemented by the *Betsy Service* which receives the required deploy data or the location of the log files and interacts with the engine on this basis. This interaction with the engines is implemented the same way as in the local approach because the *Betsy Service* reuses both the deployment and collect routines from *betsy* to avoid code duplication. *vbetsy* calls the *Betsy Service* by exchanging serialized Java objects over a TCP connection.

C. Provisioning of BPEL engines

To create the required VMs for each BPEL engine under test, we need to create a) a VM image, b) install the BPEL engine under test as well as the *Betsy Service* and c) export the image as a portable VM. To achieve this, we adopt methods from the devops movement, i.e., we convert the creation of the VMs including the BPEL engines to code.

First, a minimal linux machine, the base image, is installed within a VM as our foundation for all subsequent steps. It is granted 4 GB RAM and a single core processor. Moreover, the audio capabilities are deactivated and the network is configured using a NAT adapter. This base image[15] is built upon an Ubuntu Server 12.04.2 LTS. It requires the latest system updates, a user with sudo privileges and a ssh server.

Second, the BPEL engine and the *Betsy Service* needs to be provisioned on the base image. Figure 6 shows the dependency graph of all BPEL engines (gray background), the *Betsy Service* (black background) and other software products (white background). Each edge is a *depends on* relation, e.g., Apache ODE 1.3.6 depends on Tomcat 7.0.26. This dependency graph has been converted to executable provisioning scripts that can install any node alongside its direct as well as transitive dependencies. These provisioning scripts are implemented with *sprinkle* v0.7.6.2. The domain specific language of *sprinkle* allows to declare such a dependency graph straight-forward in its own executable constructs. Each node in Figure 6 is implemented as a *sprinkle* package which includes the download, installation, configuration, and start of the component in terms of a sequence of command line calls. The edges between the nodes are directly ported to dependencies between these sprinkle packages. Based on these packages, we created so called *sprinkle* policies for each VM which automatically install both the engine as well as the *Betsy Service*. The *sprinkle* scripts containing all packages and policies are open source and publicly available at https://github.com/uniba-dsg/betsy-engines. Next, we provisioned all VMs by applying each of the six policies on a separate base VM.

Third, the six provisioned VM images have been exported to portable `ova` archives which are available for download at https://lspi.wiai.uni-bamberg.de/svn/betsy/ova. During the execution of *vbetsy*, these `ova` archives are downloaded from the given URL, imported into VirtualBox, and started. However, these host independent and therefore portable `ova` packages cannot contain host dependent snapshots. To circumvent this problem, the snapshots are created on demand by *vbetsy* itself. This is possible as the BPEL engines and the *Betsy Service* are configured to start automatically during system startup. Thus, *vbetsy* downloads the vm, imports it into VirtualBox and starts it. When the engine and the *Betsy Service* are up and running, it saves a snapshot which is then reused for any subsequent tests.

To make this process repeatable, we automated it. As the base VM only has to be created once, this step can be automated by reusing a previously created base VM. For the other two steps, we implemented a Groovy script[16] which can automatically provision a VM for any of the six BPEL engines and export the image as an `ova` archive. For engines that require manual installation steps, the archive can also be created manually.

D. Issues

During the development of *vbetsy*, we faced two major issues regarding the usage of VirtualBox. First, as we automatically apply port forwarding rules to our VMs via the Java API of VirtualBox, we detected a bug[17] within VirtualBox v4.2.12. The bug was hard to track down as it occurred non-deterministically only on Mac OS X. Second, every machine

[15]The base image is available at https://lspi.wiai.uni-bamberg.de/svn/betsy/ova/basevm.ova.

[16]The script is available at https://github.com/uniba-dsg/betsy/blob/master/src/main/groovy/betsy/tool/VirtualMachineInstaller.groovy

[17]See https://www.virtualbox.org/ticket/11635 for the bug report.

Table I
AVERAGE EXECUTION TIME IN SECONDS OF THE ENGINE LIFECYCLE TASKS PER ENGINE BEFORE AND AFTER USING VIRTUALIZATION.

	before / local				after / virtual				diff / local - virtual			
Engine	install	start	stop	Σ	install	start	stop	Σ	install	start	stop	Σ
ActiveBPEL v5.0.2	14.12s	5.02s	0.52s	19.66s	0.30s	1.86s	0.58s	2.74s	13.82s	3.16s	-0.05s	16.92s
bpel-g v5.3	3.74s	9.04s	0.47s	13.25s	0.33s	1.72s	0.47s	2.52s	3.42s	7.32s	0.00s	10.73s
Apache ODE v1.3.5	5.67s	9.30s	0.47s	15.43s	0.25s	1.70s	0.23s	2.17s	5.42s	7.61s	0.23s	13.26s
OpenESB BPEL SE v2.2	129.11s	26.52s	7.96s	163.59s	0.21s	1.82s	0.46s	2.49s	128.90s	24.70s	7.50s	161.10s
Orchestra v4.9.0	18.06s	12.37s	0.49s	30.91s	0.22s	1.93s	0.23s	2.38s	17.83s	10.44s	0.26s	28.53s
Petals BPEL SE v4.0	7.05s	19.43s	0.48s	26.96s	0.25s	1.82s	0.28s	2.34s	6.81s	17.61s	0.20s	24.61s
average	29.63s	13.61s	1.73s	44.97s	0.26s	1.81s	0.37s	2.44s	29.37s	11.80s	1.36s	42.53s
min	3.74s	5.02s	0.47s	13.25s	0.21s	1.70s	0.23s	2.17s	3.42s	3.32s	0.24s	11.08s
max	129.11s	26.52s	7.96s	163.59s	0.33s	1.93s	0.58s	2.74s	128.90s	24.59s	7.38s	160.85s
standard deviation	49.04s	7.94s	3.05s	58.50s	0.04s	0.09s	0.15s	0.19s				

in VirtualBox is configured with a `LinkUpDelay` by default, which ensures that packets frozen in the network stack of a snapshot will not get lost when the network environment changes. To achieve this, the network stack is enabled with a delay of approx. five seconds, during which all sent packets to this machine are lost. Instead of waiting for five seconds for the machine to become available, we disabled this option.

V. EVALUATION

In this section, we present the evaluation of the feasibility of our approach and tool by determining the effects on a) install as well as start time, our main objective, and b) test time, i.e., any side effects due to the virtualization overhead. To prove that we can reduce both install and start time significantly, we execute the same conformance tests five times for the two variants: *betsy* and *vbetsy*, or in other words, with and without virtualization. The machine, on which the experiment is conducted, has an Intel i7-2600 processor, 16GB of RAM, and a Western Digital WD10EALX hard drive. Software-wise, it is equipped with Windows 7 64 bit, the Java 7u45, *soapUI* 4.6.2, and VirtualBox 4.2.16 which are required by *betsy* and *vbetsy*, respectively. In addition, we set up a continuous integration server to help us orchestrate the execution of the different test runs as well as gather the produced results. For that purpose, we installed and configured the continuous integration server Jenkins CI v1.545 along with Git for Windows v1.8.5.2.

The results of the experiment regarding the engine life cycle tasks install, start and stop are shown in Table I. Each row gives the durations of the three tasks as well as their sum for all six engines executed with or without virtualization (virtual vs. local). These values are given in seconds and are the average of three values from the five runs as we discard both the highest and the lowest value to reduce inaccuracy of measurement. These values are stable enough for this evaluation as the relative standard deviation is below 19% for 70 of our 72 values. While the two other values have a higher relative standard deviation, they are below one second, hence, do not affect the overall results. The last column contains the difference between both variants in seconds, denoting the improvements made by our approach. Moreover, the averages, min as well as max values, and standard deviations per task over all engines are presented as well.

Before using virtualization techniques, *betsy* relied only on local installation, startup and shutdown procedures. The six engines can be put into three groups according to their total life cycle time. bpel-g (13.25s), Apache ODE (15.43s) and ActiveBPEL (19.66) lead the field with at most 20 seconds, followed by both Petals ESB (26.96s) and Orchestra (30.91s) which approx. thirty seconds. OpenESB (163.59s) comes in last as it requires almost three minutes on our test machine. These numbers reflect the complexity of the runtime container of the engines, as the ones of the top group, namely Apache ODE, bpel-g, and ActiveBPEL, run on a light-weight servlet container, while the engine that came in last, the BPEL engine of the OpenESB, is running within an ESB that is deployed onto a heavy-weight application server. A light-weight container, however, does not guarantee fast install, start and stop times as Orchestra shows, while a BPEL engine within an ESB does not necessarily lead to long install, start and stop times. What is more, the major impact factor of the total time varies from engine to engine. For OpenESB, Orchestra and ActiveBPEL, the install time is the driving factor, while for the other three engines, the start time is higher than the install and stop time. Moreover, the time to stop an engine can almost be neglected as it is .5 seconds for all engines, except for OpenESB which requires approx. eight seconds to shutdown. This is because OpenESB is stopped gracefully via a shutdown method while the OS processes of the other engines are simply killed. In the min, max and standard deviation values, the wide range of these engine dependent durations can be observed. The install time has a range of approx. 125 seconds, followed by the range of the start task which accounts to approx. 25 seconds. The stop task has the lowest range with approx. seven seconds. For the install and the stop task, the standard deviation is higher than the average, showing that there are very high differences in the base values, while the start task is quite balanced around 30 seconds.

Table II
AVERAGE EXECUTION TIME IN SECONDS OF THE ENGINE ACTIONS AND THE TEST TASK PER ENGINE BEFORE AND AFTER USING VIRTUALIZATION

	before / local				after / virtual				diff / local - virtual			
Engine	deploy	test	collect	Σ	deploy	test	collect	Σ	deploy	test	collect	Σ
ActiveBPEL v5.0.2	18.16s	0.79 s	0.02s	18.97s	20.18s	0.83 s	0.19s	21.20s	-2.03s	-0.04s	-0.16s	-2.23s
bpel-g v5.3	9.52s	0.86 s	0.02s	10.40s	9.01s	0.93 s	0.18s	10.12s	0.50s	-0.07s	-0.16s	0.27s
Apache ODE v1.3.5	3.53s	1.52 s	0.05s	5.10s	4.01s	1.92 s	0.13s	6.06s	-0.48s	-0.39s	-0.09s	-0.96s
OpenESB BPEL SE v2.2	4.04s	0.84 s	0.06s	4.93s	7.97s	0.85 s	0.17s	8.99s	-3.93s	-0.02s	-0.11s	-4.05s
Orchestra v4.9.0	2.53s	0.77 s	0.02s	3.33s	7.30s	0.96 s	0.20s	8.46s	-4.76s	-0.19s	-0.17s	-5.13s
Petals BPEL SE v4.0	7.05s	2.32 s	0.03s	9.41s	9.66s	3.38 s	0.12s	13.16s	-2.61s	-1.06s	-0.08s	-3.75s
average	7.47s	1.18s	0.04s	8.69s	9.69s	1.48s	0.16s	11.33s	-2.22s	-0.30s	-0.13s	-2.64s
min	2.53s	0.77s	0.02s	3.33s	4.01s	0.83s	0.12s	6.06s	-4.76s	-0.06s	-0.09s	-2.73s
max	18.16s	2.32s	0.06s	18.97s	20.18s	3.38s	0.20s	21.20s	0.50s	-1.06s	-0.14s	-2.23s
standard deviation	5.84s	0.63s	0.02s	5.74s	5.51s	1.02s	0.03s	5.36s				

Looking at the numbers after applying our extension to *betsy* with virtualization techniques, the picture changes dramatically. The install time ranges between .21s and .33s, the start time between 1.70s and 1.93s and the stop time between .23s and .58s. All engines now have almost the same duration regardless of their previous durations. Instead of having wide ranges, the durations can be seen as constants. Thus, the highly engine dependent durations have been converted to engine independent values. The highest gain is achieved for OpenESB as it saves 161.10 seconds to create and throw away a fresh instance, while the savings for the other engines range from approx. 10s up to 28s.

Next, we investigate the possible side effects of this approach on the other engine-related steps of the *betsy* testing process, namely, on the deploy, test and collect step. In Table II, the timings of the test step are shown analogous to the previous table. Orchestra, OpenESB and Apache ODE have the fastest deployment process lasting at most four seconds. Petals and bpel-g form the group in the middle with 7.05s and 9.52s while ActiveBPEL comes in last with 18.16s. The durations of the test task do not vary as much as the deploy task, ranging only from .77s to 2.32s. The collect task is executed very fast as it copies files from one folder to another on a single hard drive.

Looking at the right part of Table II, we can see a clear overhead. All durations, except for a single value[18], have increased. The collect task takes .13 seconds longer on average and the test task .3 seconds. The largest effect of our extension of *betsy* on the duration of these tasks is seen for the deploy task as deployment lasts 2.22 seconds longer. For the collect task, the changes are completely neglectable whereas both the deploy and test task have increased up to almost five seconds in a single case. When comparing the increase in time of these three steps with the decrease in time regarding the install, start and stop tasks, we can say that in that context, they are neglectable as we can still save at least eleven and at most 157 seconds for executing a single test case. In terms of percentages, we were able to reduce the test case execution time including the *setup*, *test execution* and *teardown* phases dramatically as shown in Table III. Improvements range between 38% for ActiveBPEL and 93% for OpenESB, whereas the other four engines have a reduction of at least 47% and at most 68%.

[18]We account this to the fact that bpel-g performs faster on Ubuntu in the VM as on Windows on the host.

Table III
OVERALL REDUCTION IN TEST CASE EXECUTION TIME.

	Act.BPEL	bpel-g	ODE	OpenESB	Orch.	Petals
Δ	14,69s 38%	11,01s 47%	12,30s 60%	157,05s 93%	23,40s 68%	20,86s 57%

To sum up, this answers the research question, as our experiment shows that it is possible to create fresh and started instances of such software in a timely fashion independent of any complex installation or startup procedure using virtualization techniques. Furthermore, it dramatically reduces overall execution time despite the additional virtualization overhead.

VI. LIMITATIONS

As *vbetsy* in contrast to *betsy* decreases execution time at the expense of space, this approach requires more RAM and disk space. While the additional disk space and RAM for using *VirtualBox* is neglectable, the VMs themselves introduce a major overhead due to the operating system (OS) in the VM onto which the engines are installed. As RAM is more scarce then disk space, RAM may become a bottleneck, especially when testing multiple engines at the same time. Replacing the hard disks used in the evaluation with solid state disks (SSDs) will reduce the durations for *betsy* and *vbetsy*, as both rely heavily on I/O throughput. However, it is currently unknown whether it increases or decreases the advantages of *vbetsy* over *betsy*. The additional overhead of the OS of the VM is also influencing the latency for any communication with the engine under test, thus, causing an increase in latency for the actual test step. As this is inherent to our approach, this has to be taken as a given.

The approach can use any VM image bundled as a portable `ova` file, however, the system that imports and uses this image must be able to fulfill the hardware requirements for the image, thus, the portable image has a minimum hardware dependency. Moreover, the snapshots themselves are not portable and have to created per machine. Consequently, upon updating the VM image, any snapshot based on this VM has to be recreated, causing additional overhead. Alternatively, a newer snapshot can be created upon an existing one. But this evolution strategy can only be used on single machines as the snapshots are system-dependent.

The approach is a good fit for the cloud and its available infrastructure as a service (IaaS) products because new instances of machines can be easily spawned and discarded. However, major IaaS vendors, e.g., *Amazon EC2*, solely support the creation of disk but no RAM snapshots. Hence, only a part of our approach is directly implementable on such IaaS systems. In addition, our approach does not take security into account, which would be necessary when leveraging such IaaS offerings.

VII. Conclusion and Future Work

In this paper, we presented an approach to provide fresh and started instances of complex software in an effective and efficient manor to support test isolation and automation. By evaluating the install, startup and shutdown times of six BPEL engines with and without our approach, we showed that we are able to convert the software dependent install, startup and shutdown times to constants. Nevertheless, there is an increase in latency due to the virtualization overhead for executing the actual test. However, this increase is neglectable as the gain during the *setup* and *teardown* phases of the test more than outweighed the loss in the *test execution* phase, leading to time savings ranging from approx. 11s (38%) up to 157s (93%) in total

Future work comprises three aspects: i) reducing the overhead, ii) creating a more generic approach and iii) increasing reusability of the provisioning scripts. Firstly, using VMs inherently reduces performance due to the additional overhead of two OSs. We aim to reduce this overhead by moving from OS-based to container-based virtualization allowing to host multiple isolated containers on top the host OS. Secondly, our approach and tool can only be used in conjunction with *betsy* as it is an extension of it. To achieve separation of concerns, we want to extract a more generic tool that solely handles test setup and teardown with virtual machines. Thirdly, the provisioning of the six open source BPEL engines is currently encoded within *sprinkle* scripts and not easily reusable. We aim to increase reusability by converting our provisioning scripts to reusable TOSCA [18] artifacts to be stored in public TOSCA type repositories.

References

[1] S. Vinoski, “The performance presumption [middleware evaluation],” *Internet Computing, IEEE*, vol. 7, no. 2, pp. 88–90, 2003.

[2] A. González, E. Piel, and H.-G. Gross, “A model for the measurement of the runtime testability of component-based systems,” in *ICSTW*, 2009.

[3] A. Colyer, G. Blair, and A. Rashid, “Managing complexity in middleware,” in *Proc. 2nd AOSD ACP4IS, Boston*, 2003, pp. 21–26.

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

[5] C. Peltz, “Web Services Orchestration and Choreography,” *Computer*, vol. 36, no. 10, pp. 46–52, October 2003.

[6] S. Harrer, J. Lenhard, and G. Wirtz, “Open Source versus Proprietary Software in Service-Orientation: The Case of BPEL Engines,” in *ICSOC*, 2013.

[7] S. Harrer and J. Lenhard, “Betsy–A BPEL Engine Test System,” Otto-Friedrich Universität Bamberg, Tech. Rep. 90, July 2012.

[8] J. Lenhard, S. Harrer, and G. Wirtz, “Measuring the Installability of Service Orchestrations Using the SQuaRE Method,” in *SOCA*, 2013.

[9] S. Harrer, J. Lenhard, and G. Wirtz, “BPEL Conformance in Open Source Engines,” in *Proceedings of the 5th IEEE SOCA’12, Taipei, Taiwan*. IEEE, 2012.

[10] H. Koziolek, “Performance evaluation of component-based software systems: A survey,” *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[11] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann, “Performance evaluation of message-oriented middleware using the specjms2007 benchmark,” *Performance Evaluation*, vol. 66, no. 8, pp. 410–434, 2009.

[12] S. Strauch *et al.*, “Implementation and Evaluation of a Multitenant Open-Source ESB,” in *ESOCC*, 2013.

[13] Bianculli *et al.*, “SOABench: Performance Evaluation of Service-oriented Middleware Made Easy,” in *ICSE*, 2010.

[14] ——, “Automated Performance Assessment for Service-oriented Middleware: A Case Study on BPEL Engines,” in *WWW*, 2010.

[15] D. Roller, “Throughput improvements for bpel engines : implementation techniques and measurements applied to swom,” Ph.D. dissertation, IAAS, Stuttgart, Germany, 2013.

[16] G. Hackmann *et al.*, “Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices,” in *ICSOC*, 2006.

[17] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, “Workflow Patterns,” *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, July 2003.

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

[19] O. Kopp *et al.*, “Winery - A Modeling Tool for TOSCA-based Cloud Applications,” in *ICSOC*, 2013.

[20] T. Binz *et al.*, “OpenTOSCA - A Runtime for TOSCA-based Cloud Applications,” in *ICSOC*. Springer Berlin Heidelberg, 2013.

VIII. Demonstration Plan

In our live demonstration of *vbetsy* we will show the performance difference with and without our contribution of using virtual machines with snapshots instead of reinstalling and starting engines locally. This is done in three subsequent steps, namely, (A) execute one test for two open source engines with betsy, (B) execute one test for two open source engines with *vbetsy*, and, (C) show the comparison of the time differences.

A. Execute Test with betsy (without contribution), approx. 5 min

In the first step, the conformance testing tool *betsy* is executed. In this run, we test the Apache ODE 1.3.5 and the OpenESB v2.3 engine. We have chosen the test case Sequence as it tests the simplest BPEL process which only contains a `receive` and a corresponding `reply` activity that echos the received message back to the caller.

During the time *betsy* is executing, we will show the log file `betsy_time.log` via the `$ tail -f` command to understand the execution flow and get a first glimpse at the data we will compare in step C.

```
$ betsy ode,openesb Sequence
```

B. Execute Test with vbestsy (with contribution), approx. 3 min

In step B, *vbetsy* is executed. In technical terms, we invoke *betsy* with the same parameters as in step (A), except we use the previously created virtual machines instead of the local BPEL engines. To achieve this, we have to add the suffix `_v` for each engine name.

```
$ betsy ode_v,openesb_v Sequence
```

C. Show Time Differences (improvements), approx. 10 min

After both executions, we will compare the log files for both *betsy* and *vbetsy* which are named `betsy_time.log`. The comparison itself is done by using a side-by-side diff highlighting the differences in time. In addition, we will import the time data into excel for a more detailed analysis. The method of time measurement is the same for both *betsy* and *vbetsy*, thus, these values are comparable in that regard. The logs contain time durations for all major steps as well as their aggregates. We will drill down from the total time to verify that both install and startup time have reduced dramatically, showing that we have achieved our research goal. The comparison is done for both BPEL engines, showing also some differences when having only lightweight Tomcat as a container (Apache ODE) or being embedded in an ESB that run on Glassfish v2.2 (BPEL Service Engine in OpenESB v2.2). We will conclude our demo with a discussion about other durations, e.g., the test duration, and whether any effects of the virtualization overhead can be noticed.