Institute of Architecture of Application Systems

Replicability of Dynamically Provisioned Scientific Experiments

Karolina Vukojevic-Haupt, Florian Haupt, Dimka Karastoyanova, and Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{vukojevic, haupt, karastoyanova, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{INPROC-2014-77,
  author    = {Karolina Vukojevic-Haupt and Florian Haupt and
               Dimka Karastoyanova and Frank Leymann},
  title     = {Replicability of Dynamically Provisioned Scientific
               Experiments},
  booktitle = {Proceedings of the 7th IEEE IEEE International Conference on
               Service Oriented Computing & Applications, SOCA 2014,
               17. – 19. November 2014, Matsue, Japan},
  year      = {2014},
  pages     = {119 - 124},
  doi       = {10.1109/SOCA.2014.54},
  publisher = {IEEE}
}
```


Universität Stuttgart
Germany

Replicability of Dynamically Provisioned Scientific Experiments

Karolina Vukojevic-Haupt, Florian Haupt, Dimka Karastoyanova, and Frank Leymann
Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

***Abstract*— The ability to repeat an experiment, known as replicability, is a basic concept of scientific research and also an important aspect in the field of eScience. The principles of Service Oriented Computing (SOC) and Cloud Computing, both based on high runtime dynamicity, are more and more adopted in the eScience domain. Simulation experiments exploiting these principles introduce significant challenges with respect to replicability. Current research activities mainly focus on how to exploit SOC and Cloud for eScience, while the aspect of replicability for such experiments is still an open issue. In this paper we define a general method to identify points of dynamicity in simulation experiments and to handle them in order to enable replicability. We systematically examine different types of service binding strategies, the main source of dynamicity, and derive a method and corresponding architecture to handle this dynamicity with respect to replicability. Our work enables scientists to perform simulation experiments that exploit the dynamicity and flexibility of SOC and Cloud Computing but still are repeatable.**

Keywords— replicability; SOC; SOA; Cloud; on-demand provisioning and deprovisioning; eScience;

I. INTRODUCTION

Service oriented architectures are well established as a suitable approach for building huge, complex and massively distributed systems that are nevertheless reliable, manageable and flexible. The underlying paradigm of *service oriented computing (SOC)* defines services as the basic building blocks of such architectures. A service encapsulates certain functionality, provides it over a unified interface and is loosely coupled with other services [2]. The concept of *cloud computing* builds on the foundations of SOC. Cloud computing provides IT resources as services having well defined properties such as elasticity or pay-as-you-go [9]. The use of cloud services eases the provisioning of and the access to IT resources and allows for far more dynamic approaches for the provisioning and management of complex software systems.

The field of eScience comprises a variety of research disciplines where scientific progress depends on the use of IT [8]. One of these disciplines is simulation technology, where real world phenomena are formally modeled and then simulated based on these models. Experiments based on simulations are typically complex processes incorporating different activities, data, and tools. The use of process modeling languages and workflow systems is a common approach to manage the complexity of such experiments. In our previous work, we have developed a scientific workflow management system (SWfMS) based on conventional workflow technology and SOC but especially extended and adapted to the needs of eScience [10][5]. In our current work, we are taking this work one step further by not only exploiting the benefits of SOC but also adopting the principles of cloud computing. We introduced an approach and architecture for the on-demand provisioning and deprovisioning of workflow execution middleware and services including their underlying middleware and infrastructure [1].

Scientific progress is based on the concept of validation and reuse of research results. In our work we focus on experiments, especially "virtual" experiments conducted based on simulations. The validation of results obtained by an experiment depends on the ability to reproduce these results [6]. The term *reproducibility* in general refers to the ability to obtain the result of an experiment again, either by conducting the same experiment again (possibly in a different environment) or by following a completely different approach [11][12]. While the term reproducibility mostly focuses on the validity of the result of an experiment, in many cases the validity of the experiment itself, i.e. the method and process it is based on, is also of interest. In this context, the more specific concept of *replicability* comes into play [7], it describes the ability to conduct an experiment again resulting in the same outcome.

The application of the principles of SOC and cloud computing in the domain of eScience provide a multitude of benefits like automation, flexibility and better integration support. Such workflow-based experiments typically show a high degree of runtime dynamicity by for example dynamic service selection or resource provisioning. This altogether leads to significant challenges with respect to replicability. In this paper we therefore contribute (1) a general concept for replicability in workflow-based experiments, (2) a method to realize this replicability, (3) an extension of our architecture that supports replicability, and (4) an example of a realization of our approach.

The rest of this paper is structured as follows. In section II we introduce our general concept for replicability in workflow-based and service-oriented scientific experiments. In section III we refine this concept by linking it to service binding strategies. In section IV we show how we adapted our existing architecture to support replicability. Section V gives an example showing some details of our approach. We give an overview about related work in section VI and close the paper with a short summary and outlook in section VII.

II. Concept of Replicability

In this work we focus on the replicability of workflow-based and service-oriented experiments. The execution of an experiment is modeled as a workflow comprising a set of activities together with control flow and dataflow dependencies. The activities of a workflow model are described in terms of functional and non-functional requirements. For each activity the modeler of a workflow defines the required abstract service interface as well as additional non-functional requirements. Following the principles of SOC, the functionality (or interface) required by an activity is provided by services.

The execution of a workflow-based experiment, depicted in Fig. 1 on the left side, is based on a workflow model describing the course of the experiment, input data (X) and an execution context (A). The execution context contains the non-functional requirements for the execution of the workflow model, for example cost or time constraints. When starting an experiment, the workflow model, input data and the execution context are passed to an execution environment (E). This environment is inter alia responsible for the fulfillment of the non-functional requirements during the execution of an experiment. The execution of an experiment finally results in output data (Y).

The workflow model does not need to specify any specific services to use; it only defines the needed functionality and non-functional requirements. The selection of a suitable service is realized at runtime by the execution environment. In SOC this concept is well-known as *publish-find-bind* [2]. A service provider publishes services to a service registry ("publish"). A service consumer queries this service registry for a service fulfilling his requirements ("find") and then binds to the selected service ("bind"). To achieve loose coupling between service consumer and service provider, the find and bind steps are typically carried out at runtime by the execution environment.

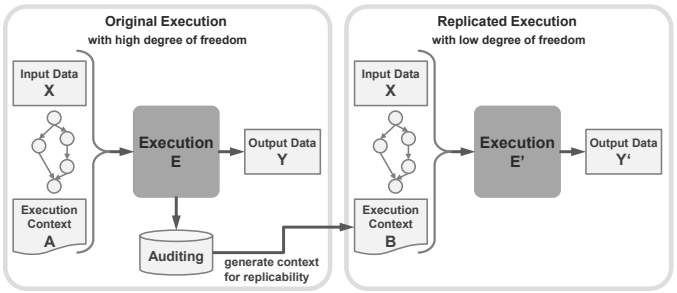

Fig. 1. Concept of replicability for scientific experiments.

During the execution of a workflow, this so called dynamic binding results in degrees of freedom for the selection of appropriate services. A tangible example is an activity requiring a web search. Google (www.google.com) and Bing (www.bing.com) both provide this functionality (web search based on a given string) with the same non-functional capabilities (free of cost and fast). The decision which of these web search services is selected at runtime is left to the execution environment. As a consequence, the replicability of workflow-based and service-oriented experiments is limited. For each execution the execution environment might select a different set of services fulfilling the functional and non-functional requirements of the workflow model and execution context.

Our approach for the replicability of workflow-based experiments is based on the principle depicted in Fig. 1. During the execution of an experiment, shown in the left part, we observe the execution of the original experiment and capture specific execution data in an auditing system. To achieve replicability, the original execution context A is replaced by a new execution context B (Fig. 1, right part) generated based on the auditing data captured before. It defines very strict requirements and thereby reduces the degree of freedom during runtime in a way that further executions of the experiment are replications of the original execution. The execution E' based on the execution context B then results in output data Y'.

The replication of a workflow-based experiment does not necessarily imply the complete replication of every aspect of an experiment. Due to the abstraction levels introduced by SOC we can in general not control every aspect of the execution of an experiment and the involved services. If for example the hardware a specific service is running on is changed or reconfigured, this might not affect the functional and non-functional capabilities declared by this service at all. Nevertheless, this change might have an impact on experiments using this service, for example by introducing a slightly different timing behavior. When talking about replication, we therefore aim at a repeated execution "as similar as possible" to the original execution.

III. Replicability and Service Binding Strategies

As already discussed in the last section, the main source of dynamicity is in the service binding at runtime. In our previous work we have developed an extended classification for service binding strategies [1]. In the following we will show how to achieve replicability for each of these classes. Our method is based on that we first identify the dynamic aspects for each binding strategy. Subsequently we show which information are needed to replicate a service call.

In Fig. 2 the different binding strategies are illustrated. A service composition is shown as an example implementation of an experiment. The service calls are passed to the so-called enterprise service bus (ESB) [3], a middleware component that realizes the different binding strategies.

When using the strategy *static binding* (Fig. 2, A), the ESB receives a service call already including a target address, the so-called *service endpoint*. The service call is directly forwarded to the given address. Static binding is the only binding strategy without a dynamic component. Service calls using the static binding strategy are readily replicable.

In case of *dynamic binding* (Fig. 2, B), the target of a service call is determined dynamically at runtime based on functional and non-functional requirements. The ESB carries out a service discovery and service selection finally resulting in a service fulfilling both, the functional requirements as well as the non-functional requirements. The service call is then forwarded to the endpoint of the selected service. To replicate a service call with dynamic binding, during repeated execution the ESB has to forward the service call exactly to the endpoint selected in the original service call. This can be achieved by the ESB logging the selected endpoint during the original execution. For a further execution no service discovery and service

selection are performed, instead the before logged service endpoint is selected.

Using the strategy *dynamic binding with service deployment* (Fig. 2, C), the required service will be initially deployed before it can be used. First the ESB receives the service call and carries out a service discovery and service selection. The difference is that the ESB does not get an endpoint but a service implementation of a suitable service. The service implementation is then deployed on an existing middleware and then the service call is passed to its service endpoint. To replicate such a service call during repeated execution the ESB has to deploy exactly the same service implementation on exactly the same middleware as in the original execution. This can be achieved by the ESB logging which service implementation was deployed on which middleware. For repeated execution the service discovery and service selection are skipped. Instead the service implementation logged before is deployed on the middleware also logged before.

In the case of *dynamic binding with software stack provisioning* (Fig. 2, D) not only the service implementation but also the underlying middleware and infrastructure are provisioned before the service call can be forwarded. First the ESB receives the service call and carries out a service discovery and a service selection. As result the ESB obtains a so-called *service package*, an archive containing all required artifacts to provision the service implementation including its underlying middleware and infrastructure. The service package is then provisioned in an existing cloud infrastructure and the service call is forwarded to its service endpoint. To replicate such a service call, during repeated execution the ESB has to provision exactly the same service package in exactly the same cloud infrastructure as in the original execution. This is achieved by the ESB logging which service package is provisioned in which cloud infrastructure. During repeated execution the ESB uses exactly the service package logged before and provisions it on the cloud infrastructure also logged before using exactly the same parameters as in the original execution.

We summarize our method in Table 1. For "static binding" there are no dynamic aspects and therefore replicability is given without any further action. For "dynamic binding" the service endpoint is determined dynamically and therefore has to be logged for repeatable execution. For "dynamic binding with service deployment" middleware and service implementation are determined dynamically. Consequently, the service endpoint is also dynamic. For repeatable execution the same middleware and the same service implementation have to be used and therefore be logged. In case the deployment of the service implementation on the middleware can be parameterized, these parameters also have to be logged. The service endpoint depends on the service deployment and is therefore not logged. For "dynamic binding with software stack provisioning" the cloud infrastructure as well as the service package is determined dynamically. Therefore the service endpoint is also dynamic for this binding strategy. For repeatable execution the cloud infrastructure and the service package as well as the provisioning parameters have to be logged. The service endpoint is not logged as it depends on the provisioning of the service package.

TABLE I. METHOD FOR REPLICABILITY

binding strategy	dynamic aspects	replicability needs
static binding	-	-
dynamic binding	service endpoint	service endpoint
dynamic binding with service deployment	middleware, service implementation, service endpoint	middleware, service implementation, deployment parameters
dynamic binding with software stack provisioning	cloud infrastructure, service stack (infrastructure, middleware, service implementation), service endpoint	cloud infrastructure, service package, deployment parameters

We assume that the used services, service implementations, service packages and cloud infrastructures are in principle available for repeated execution. In practice, it is quite possible that this assumption is violated and e.g. a certain service endpoint is no more available for later execution. In this case a replication of the original execution is not possible. The execution of the workflow would be canceled at this point. Nevertheless, to ensure a robust workflow execution, the strength of dynamic binding can be exploited to select an alternative service with the same functional and non-functional properties. It is important to log this deviation of the replication and to

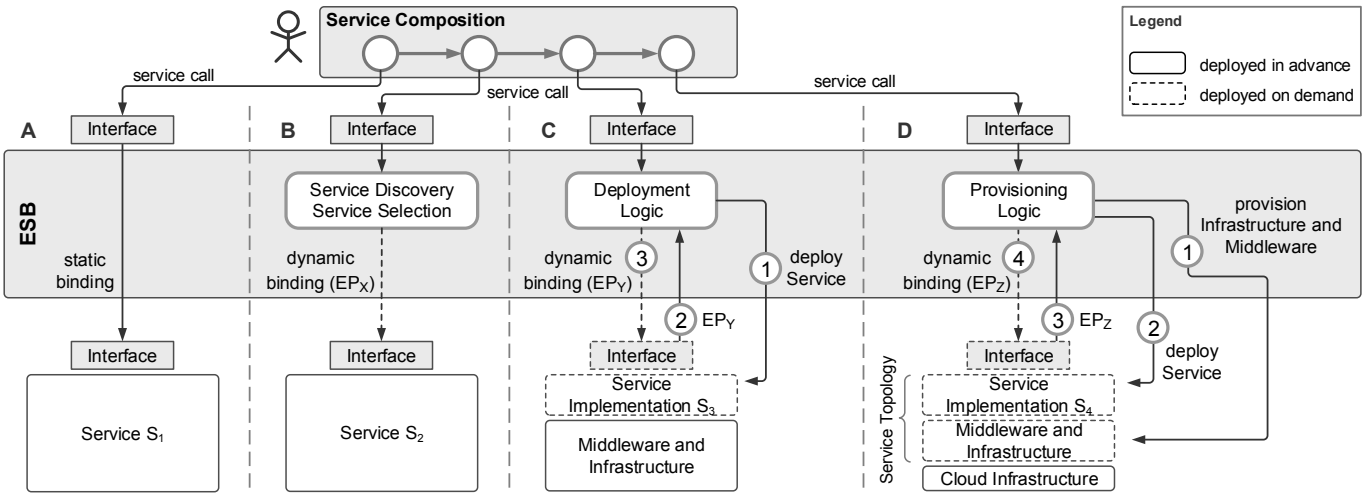

Fig. 2. Extended classification of service binding strategies, based on [1]

communicate it to the user. He can then decide if this repeated execution is usable despite the deviation.

In our previous work we have developed the concept of the on-demand provisioning of workflow execution middleware and services [1], which is based on the binding strategy "dynamic binding with software stack provisioning" (Fig. 2, D).

IV. Architecture for Replicable Experiments

In the following we will show how we realized the general concept of replicability for scientific experiments introduced in section II in our existing architecture for on-demand provisioning of workflow execution middleware and services [1][4]. This architecture realizes the service binding strategy "dynamic binding with software stack provisioning" and in addition also supports "static binding" and "dynamic binding". We distinguish between "provisioned services" and "not provisioned services". A provisioned service is a functionality provided at an endpoint with certain non-functional properties, everything else is transparent. In contrast, a not provisioned service at first has to be explicitly provisioned before it can be used.

In Fig. 3 we present the part of our architecture realizing the service binding. The workflow engine is responsible for the execution of the workflows. The ESB coordinates the processing of the service calls. The service registry is a global directory containing information about all services. It offers information about functional and non-functional properties of a service. For provisioned services the endpoint is stored, while for not provisioned services a reference to the service package repository is stored. For each not provisioned service the service package repository contains the corresponding service package together with provisioning metadata. The provisioning manager is capable to provision service packages using a suitable provisioning engine.

Service calls are initiated by the workflow engine (Fig. 3, step 1) and contain the actual payload as well as different metadata (step 2). The functional requirements (FR) describe the required interface; the non-functional requirements (NFR) describe requirements concerning the quality of a service. Whereas these requirements correspond to traditional SOC concepts, the provisioning requirements (PR) are specific for our on-demand provisioning approach. They describe requirements specific for the provisioning process, for example allowed cloud providers or the region where resources have to be provisioned.

All service calls are processed by the ESB. When receiving a service call, the ESB first executes a service discovery (step 3). In this step all service offers which are compliant with the functional requirements of the service call are determined by the service registry (step 4). Afterwards a service selection is carried out (step 5). In this step all service offers fulfilling additional to the functional requirements also the non-functional requirements are determined (step 6). If the result set contains at least one provisioned service, the service selection component returns exactly one endpoint (of a provisioned service) and the ESB forwards the service call to the selected endpoint (step 7a). If the result set however contains no provisioned services, the service selection component returns a service package reference for each service offer in the result set. The ESB forwards these service package references together with the provisioning requirements to the provisioning manager (step 7b). The provisioning manager executes a service package selection by passing the set of service package references, the provisioning requirements and its own provisioning capabilities to the service package repository (step 8). In this step a service package of the provided set is determined which on the one hand fulfills the provisioning requirements of the service request and which on the other hand can be processed by one of the available provisioning engines of the provisioning manager (step 9). The provisioning manager then provisions the resulting service package using a suitable provisioning engine (step 10). In the last step the ESB forwards the service request to the service provisioned before (step 11).

In section III we discussed that, depending on the service binding strategy, different information has to be logged during the workflow execution to enable replicability. Considering the

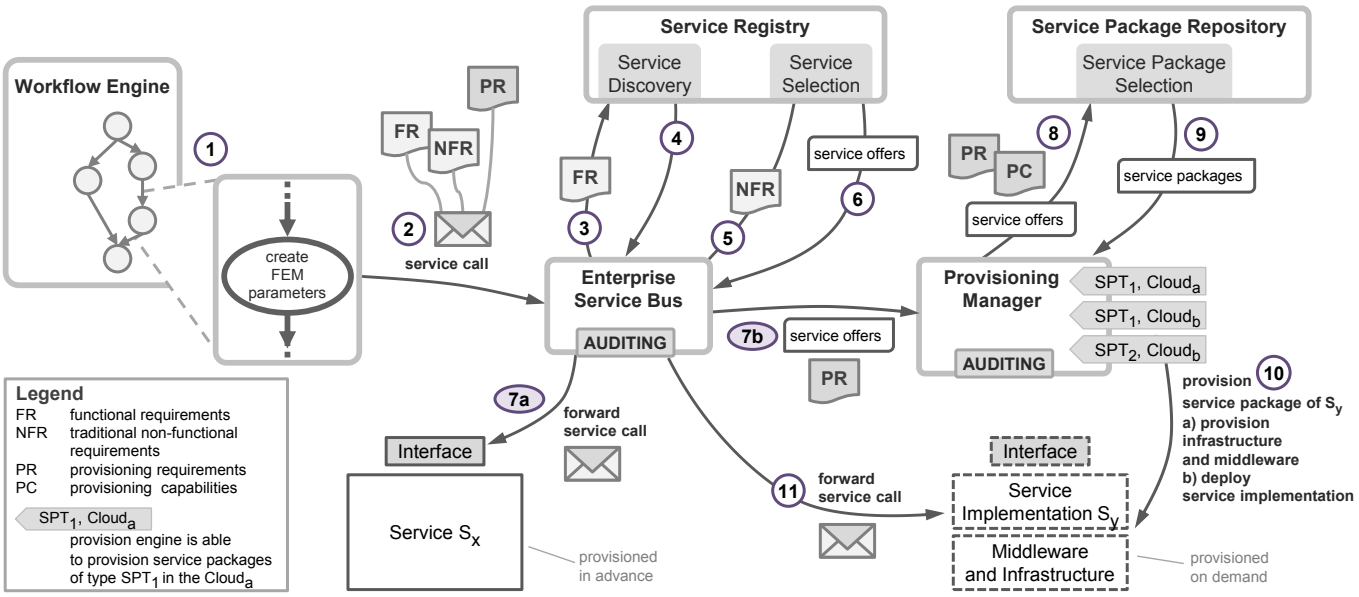

Fig. 3. Architecture for replicable experiments provisioned on demand

architecture presented before, both the ESB and the provisioning manager have to store information for replicability. For the case that the service selection step returns exactly one service endpoint, this information is logged by an auditing component in the ESB. Otherwise, an auditing component in the provisioning manager logs which service package is provisioned by which provisioning engine in which cloud environment using which parameters. Using the data captured by the auditing components, an execution context can be generated that enables the replication of the original execution.

V. Realization Example

In section II we introduced a generic concept to enable the replication of workflow-based and service-oriented scientific experiments. In the previous section we demonstrated the extension of our architecture in order to support this concept. In the following we will show, using a simplified example, how the execution context, the auditing, and the exchanged messages can be realized in a SOAP web service-based system.

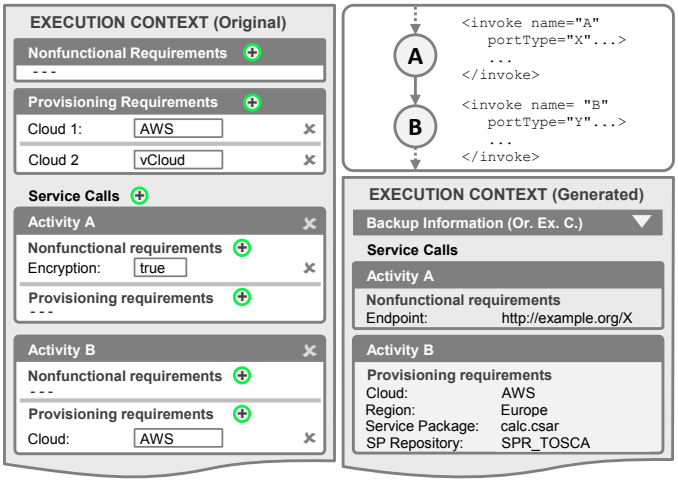

Fig. 4. Original (left) and generated execution context (right)

In the upper right of Fig. 4 a workflow model fragment containing two activities A and B is depicted. Activity A calls a service implementing interface X, activity B calls a service implementing interface Y. On the left side of Fig. 4 an execution context for this process model fragment (*original execution context*) is shown. On the one hand, the execution context allows specifying non-functional and provisioning requirements applying to the whole workflow. In our example we define that AWS or vCloud have to be used as cloud environments. The execution context also allows specifying requirements for a single activity. In our example any service used by activity A has to support encryption. For activity B it is required to use only AWS as cloud infrastructure. We assume that local requirements always supersede global requirements.

In Listing 1 and Listing 2 it is shown, which messages are sent during the execution of the workflow model fragment depicted in Fig. 4, and which data is captured in the auditing system. On the left side of Listing 1 the service call of activity A is depicted. The requirements specified in the original execution context are integrated into the header block of the message. The element <*repEx*> specifies if the service call is part of an original or part of a replicated execution. Depending on this either the original execution context or the generated execution context is considered by the execution environment.

```
<soap:Envelope>
 <soap:Header>
  <messageId>MI_102</messageId>
  <portType>X</portType>
  <repEx>no</repEx>
  <originalExecution>
   <nfr>
    <encryption>true</encryption>
   </nfr>
   <pr>
    <cloud name="AWS"/>
    <cloud name="vCloud"/>
   </pr>
  </originalExecution>
  <reproducedExecution/>
 </soap:Header>
 <soap:Body>...</soap:Body>
</soap:Envelope>
```

```
<log>
 <serviceCall timestamp="... "
              type="dynamicBinding">
  <sender>
   <pModel>PM_304</pModel>
   <pInstance>PI_12</pInstance>
   <activity>A</activity>
  </sender>
  <repEx>no</repEx>
  <portType>X</portType>
  <target>http://example.org/X</target>
    ...
 </serviceCall>
  ...
</log>
```

Listing 1. Service call of actitivity A, based on original context

On the right side of Listing 1 a consolidated log for this service call is depicted. It incorporates logging data from the workflow engine, the enterprise service bus and the provisioning manager. In our example the log shows that activity A was executed by a provisioned service with the endpoint address "http://example.org/X". This data can then be used to generate an execution context for the repeated execution.

```
<soap:Envelope>
 <soap:Header>
  <messageId>MI_103</messageId>
  <portType>Y</portType>
  <repEx>no</repEx>
  <originalExecution>
   <nfr/>
   <pr>
    <cloud name="AWS"/>
   </pr>
  </originalExecution>
  <reproducedExecution/>
 </soap:Header>
 <soap:Body>...</soap:Body>
</soap:Envelope>
```

```
<log>
 <serviceCall timestamp="..."
              type="onDemandProvisioning">
  <sender>
   <pModel>PM_304</pModel>
   <pInstance>PI_12</pInstance>
   <activity>B</activity>
  </sender>
  <repEx>no</repEx>
  <portType>Y</portType>
  <cloud>AWS</cloud>
  <servicePackage>calc.csar</servicePackage>
  <SPRep>SPR_TOSCA</SPRep>
  <target>http://ec2-176-34-81-162.eu-west-1.
          compute.amazonaws.com/calc</target>
            ...
  </serviceCall>
  ...
</log>
```

Listing 2. Service call of actitivity B, based on original context

In Listing 2 the service call for activity B is depicted. The requirements are again integrated into the header block. The consolidated log (right side of Listing 2) shows that a not provisioned service was selected. Specifically, the service package "calc.csar" was provisioned in the Europe region of AWS.

```
<soap:Envelope>
 <soap:Header>
  <messageId>MI_145</messageId>
  <portType>X</portType>
  <repEx>yes</repEx>
  <originalExecution>
   <nfr>
    <encryption>true</encryption>
   </nfr>
   <pr>
    <cloud name="AWS"/>
    <cloud name="vCloud"/>
   </pr>
  </originalExecution>
  <reproducedExecution>
   <nfr>
    <target>http://example.org/X
    </target>
   </nfr>
   <pr/>
  </reproducedExecution>
 </soap:Header>
 <soap:Body>...</soap:Body>
</soap:Envelope>
```

```
<soap:Envelope>
 <soap:Header>
  <messageId>MI_146</messageId>
  <portType>Y</portType>
  <repEx>yes</repEx>
  <originalExecution>
   <nfr/>
   <pr>
    <cloud name="AWS"/>
   </pr>
  </originalExecution>
  <reproducedExecution>
   <nfr/>
   <pr>
    <cloud name="AWS" region="Europe"/>
    <servicePackage>calc.csar</servicePackage>
    <SPRep>SPR_TOSCA</SPRep>
   </pr>
  </reproducedExecution>
 </soap:Header>
 <soap:Body>...</soap:Body>
</soap:Envelope>
```

Listing 3. Call of actitivity A (left) and B (right), based on generated context

On the lower right part of Fig. 4 the execution context generated from the auditing data captured during the original exe-

cution is depicted. For activity A the specific endpoint "http://example.org/X" is specified - thus the same service as in the previous execution should be called. For activity B the service package "calc.csar" should be used and likewise be provisioned in the Europe region of the AWS Cloud. In Listing 3 we show the resulting service calls of activity A and B during the repeated execution based on the generated execution context. The header blocks now contain the more specific requirements to ensure the same execution as the original one.

VI. Related Work

In [13] the authors show how cloud related technologies can be used to setup and run eScience software independent of the underlying cloud platform. Experiments are assumed to be simple programs or scripts; there is no notion of SOC or workflows and consequently no runtime dynamicity like dynamic binding. Replicability is mainly related to the technical setup process, an aspect that is in our work encapsulated by service packages and the related provisioning engines together with a dynamic service package selection process at runtime [4].

The work presented in [14] proposes the use of cloud based virtual machines as a means to archive, distribute and document IT-based experiments. The approach focuses mainly on single machine scenarios; there is no generic way to handle complex application architectures. In contrast to our work, the execution and coordination of complex experiments is not covered at all. Replicability is mainly seen as the replicable setup of a complex set of tools on a (virtual) machine. Our approach tackles the replication of the whole process of dynamically setting up complex application architectures and executing long running workflows incorporating extensive dynamic binding. In addition, the work presented in [14] does not exploit the power of cloud computing at all; it is solely based on the use of virtual machines without considering aspects like elasticity or pay-as-you-go.

The work presented in [15] provides an algorithm that analyzes provenance information to determine if an experiment has been replicated. Our work in contrast focuses on the runtime of experiments and aims to ensure that a running experiment is a replication of a previous execution. In addition, our approach focuses on control flow centric workflow models. The workflow execution system the work in [15] is based on realizes the binding strategy "dynamic binding with service deployment" (Fig. 2, C) while our architecture realizes the more dynamic strategy of "dynamic binding with software stack provisioning" (Fig. 2, D).

VII. Summary and Outlook

In this paper we have demonstrated how replicability can be achieved for workflow-based and service-oriented experiments. After presenting our general approach to control replicability by means of an appropriate execution context, we used an existing classification of service binding strategies to systematically derive how replicability can be realized in SOC based systems. After that, we showed how our existing architecture has to be extended to support the replication of experiments. In addition, we presented an example to give some details about a realization of the proposed solution. Our work allows scientists to benefit from the dynamicity and flexibility of SOC and cloud based experiments without losing the capability to replicate their work.

Besides the ongoing realization of the presented replicability features in our existing system the topic of "controlled dynamicity" promises further potential with respect to eScience experiments. Parameter studies are often used to run multiple simulations that differ in just one input parameter. This concept can also be transferred to the level of the execution environment. Service binding is then seen as a parameter of an experiment. This approach can be used to analyze the influence of specific services (algorithms) on the execution as well as the result of an experiment.

ACKNOWLEDGEMENT

K. Vukojevic-Haupt and D. Karastoyanova would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC310/1) at the University of Stuttgart. This work was partially funded by the BMWi project Migrate! (01ME11055).

References

[1] Vukojevic-Haupt, K.; Karastoyanova, D.; Leymann, F.: *On-demand Provisioning of Infrastructure, Middleware and Services for Simulation Workflows*. In: Proceedings of SOCA 2013

[2] Papazoglou, M.P.: *Service-oriented computing: concepts, characteristics and directions*. In: Proceedings of WISE 2003

[3] Chappell, D.: *Enterprise Service Bus: Theory in Practice*. 2004

[4] Vukojevic-Haupt, K.; Haupt, F; Karastoyanova, D.; Leymann, F.: *Service Selection for On-demand Provisioned Services*. In: Proceedings of EDOC 2014

[5] Sonntag, M.; Karastoyanova, D.: *Ad hoc Iteration and Re-execution of Activities in Workflows*. In: International Journal On Advances in Software. Vol. 5 (1 & 2), Xpert Publishing Services, 2012

[6] Leymann, F.: *Linked Compute Units and Linked Experiments: Using Topology and Orchestration Technology for Flexible Support of Scientific Applications*. In: Software service and application engineering. Springer

[7] Giles, J. *The trouble with replication*. In: news@nature 442, 7101, 344–347. 2006.

[8] Hey, T., Tansley, S., and Tolle, K. 2009. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft

[9] *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, 2011, http://www.nist.gov/itl/cloud/

[10] Görlach, K. et al.: *Conventional Workflow Technology for Scientific Simulation*. In: Guide to e-Science, Springer-Verlag, 2011.

[11] Drummond , C.: *Replicability is not reproducibility: Nor is it good science*. In Proceedings of ICML 2009

[12] Missier, P. ; Woodman, S.; Hiden, H.; Watson, P.: *Provenance and data differencing for workflow reproducibility analysis*. In: Concurrency and Computation. Practice & Experience, 2013

[13] Klinginsmith, J.; Mahoui, M.; Wu, Y.: *Towards Reproducible eScience in the Cloud*. In: Proceedings of CloudCom 2011

[14] Howe, B.: *Virtual Appliances, Cloud Computing, and Reproducible Research*. In: Computing in Science & Engineering, 14, 36-41. 2012

[15] Missier, P. et al.: *Provenance and data differencing for workflow reproducibility analysis*. In: Concurrency and Computation. Practice and Experience, 2013

All links were last followed on 02.10.2014