Service Composition for REST

Florian Haupt, Markus Fischer, Dimka Karastoyanova,
Frank Leymann, Karolina Vukojevic-Haupt

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

BIBTEX:

```
@inproceedings{INPROC-2014-33,
  author    = {Florian Haupt, Markus Fischer, Dimka Karastoyanova,
               Frank Leymann, Karolina Vukojevic-Haupt},
  title     = {Service Composition for REST},
  booktitle = {2014 IEEE 18th International Enterprise Distributed Object
               Computing Conference, EDOC 2014},
  year      = {2014},
  pages     = {110-119},
  publisher = {IEEE},
  doi       = {10.1109/EDOC.2014.24}
}
```

Service Composition for REST

Florian Haupt, Markus Fischer, Dimka Karastoyanova, Frank Leymann, Karolina Vukojevic-Haupt
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

***Abstract*—One of the key strengths of service oriented architectures, the concept of service composition to reuse and combine existing services in order to achieve new and superior functionality, promises similar advantages when applied to resources oriented architectures. The challenge in this context is how to realize service composition in compliance with the constraints defined by the REST architectural style and how to realize it in a way that it can be integrated to and benefit from existing service composition solutions. Existing approaches to REST service composition are mostly bound to the HTTP protocol and often lack a systematic methodology and a mature and standards based realization approach. In our work, we follow a comprehensible methodology by deriving the key requirements for REST service composition directly from the REST constraints and then mapping these requirements to a standard compliant extension of the BPEL composition language. We performed a general requirements analysis for REST service composition, defined a meta model for a corresponding BPEL extension, realized this extension prototypically and validated it based on a real world use case from the eScience domain. Our work provides a general methodology to enable REST service composition as well as a realization approach that enables the combined composition of WSDL and REST services in a mature and robust way.**

Keywords—REST; service composition; BPEL; eScience; simulation workflow

I. INTRODUCTION

The *Service Oriented Computing (SOC)* paradigm introduces the concept of services as encapsulated and loosely coupled components that are the basic building blocks of complex software systems [1]. A service can be characterized as some functionality offered at a certain network address. One core concept of service oriented computing is to reuse and combine existing services to realize new and possibly more complex functionality. This approach is well known as *service composition*. There exist several kinds of service composition approaches, from automatically created compositions based on semantic matchmaking [2] over declarative compositions [3] to process model based compositions [4]. One of the most widespread service compositions languages is the *Web Services Business Process Execution Language (BPEL)* [5].

Besides the success of service oriented architectures, a new paradigm for building distributed systems is gaining more and more relevance. *Representational State Transfer (REST)* is an architectural style originally defined to document the design rationale behind the architecture of the *World Wide Web (WWW)* [6]. The REST architectural style is realized by the WWW, a distributed system of interconnected documents that are meant to be consumed by humans. For quite some time, there is a noticeable movement towards applying the REST principles also to the design and realization of services, i.e. to build distributed systems of interconnected resources that are meant to be accessed by software rather than by humans. Realizing services based on the basic techniques of the WWW, mainly the HTTP protocol, seems to attract service designers and developers due to its perceived simplicity, especially when compared to service oriented architectures implemented using the WS* standards [7]. In addition to that, the rise of the cloud computing paradigm has significantly increased the need for a simple but powerful means for remote resource access. The NIST definition of Cloud Computing [8] explicitly states, that cloud services have to enable “broad network access”, i.e. they have to provide an interoperable access mechanism suitable for heterogeneous client types. In practice, this requirement is often realized by providing REST interfaces.

Considering the ongoing dissemination of REST services, the question arises if and how the concept of service composition can be adapted to resource oriented architectures based on the REST architectural style. Having a look at literature, there is apparently a true need for REST service composition. In the field of automated cloud service provisioning, there exist several approaches that define the provisioning logic of complex service topologies using process models, i.e. service compositions [9] [10]. As the provisioning of such service topologies inherently requires interacting with the interfaces of cloud services, the provisioning process consequently needs to be able to interact with REST services. Another domain that successfully applies service composition concepts is the field of eScience. The execution of scientific experiments can be simplified and automated by modeling an experiment as service composition and executing it based on workflow technology [11] [12]. In addition to the use of service compositions, REST services are also disseminated in the eScience domain [13] [14]. Consequently, in the eScience domain there is a need for REST service composition.

Service oriented architectures and resources oriented architectures are based on different paradigms. Adapting the concept of service composition defined in SOA to REST based systems therefore requires a thoroughly analysis. In this paper we contribute (1) a detailed analysis of the REST architectural style with respect to service composition. Based on that, (2) a meta model for REST service composition as an extension of the BPEL composition language is designed. We also contribute (3) a prototypical implementation of this extension and (4) validate it with a real world use case from the eScience domain.

The rest of the paper is organized as follows. Section II discusses the REST principles with respect to service composition and infers a set of corresponding requirements. In section III, the BPEL composition language is introduced and already existing solutions for REST service compositions with BPEL are shortly discussed. Section IV presents our meta model for REST service composition with BPEL together with the methodology it was designed with. The realization of the BPEL extension and the validation are shown in section Fig. 3. The paper finishes with an overview about relevant related work in section VI and conclusion and outlook in section VII.

II. Service Composition for REST

REST is an architectural style for distributed hypermedia systems [6]; its definition is based on a set of constraints. Any architecture compliant with these constraints can be called a REST (or RESTful) architecture. In this section, we will give an overview about REST, focusing mainly on the aspects relevant in the context of service composition. In addition, we will discuss for each REST constraint, which requirements relevant for REST service composition can be deduced from it.

Strictly speaking, the term “service composition” cannot be applied to REST architectures. Whereas the main entities in service oriented architectures are services providing a set of operations, in REST architectures the concept of a service does not exist at all. Instead of services, the main entities in REST are resources. The interface to interact with such resources, i.e. the operations available, is the same for all resources and predefined by the so called *uniform interface*. Whereas services differ in terms of the interface, i.e. the operations they offer, resources differ in terms of the resource representations they provide. Resources can, nevertheless, also be interpreted as services. Following this interpretation, REST services all provide the same basic operations (the uniform interface), but each “service” may provide different variants of these operations, namely differing in request and response parameters. For reasons of comprehensibility, in the following we will still use the term “REST service composition”, although we are aware, that the terms “resource composition” or “resource interaction composition” are more exact.

As starting point for the following overview about the REST principles we choose the term the acronym REST is based on, “**Re**presentational **S**tate **T**ransfer”. Although this does not cover all aspects relevant in REST, it combines two main aspects, the concept of resource representations (“Representational”) and the concept of state transfer.

Understanding the concept of representations requires to first introducing the concept of resource orientation. As already discussed before, in REST architectures all interaction is about accessing resources. The interaction with a resource, i.e. reading or writing a resource, is abstracted by the concept of representations. Each resource is available in one or more representations. When a resource is read, not the resource itself but a representation of the resource is retrieved. Just the same, when a resource is created or updated, a representation of the resource is given. The concept of resource representations can be easily illustrated using a simple example from the World Wide Web (WWW). An article of a blog is a resource available in the WWW. However, when accessing this resource, i.e. when opening the article in a web browser, not the resource itself but a HTML representation of this resource is shown. This decouples clients accessing the resources from any details how and in which format the resource is internally stored at the server. The representation(s) of a resource can be adjusted to the needs of the clients without changing the internal resource format.

The concept of resource representations introduces the need of content negotiation. As mentioned before, a resource can be provided in multiple representations. Similarly, a client accessing a resource may be able to handle some representations while it cannot handle others. Therefore, when a client accesses a resource, a matching representation supported by both, client and server, has to be returned. The process of determining such a matching representation is called *content negotiation*. There exist two types of content negotiation, *client driven content negotiation* and *server driven content negotiation*. In server driven content negotiation, the client sends a request to the server and also provides, which representations it will accept as response. The server is then responsible to select an appropriate representation to return, if available. In client driven content negotiation, the client sends a request to the server. The server then responds with a list of all available representations. In this case, the client is responsible to select an appropriate representation and can then retrieve it from the server with a second request.

To support the concept of resource representations, a REST service composition has to support content negotiation (requirement 1). More precisely, it has to be able to explicitly specify the type of representations and it has to provide means to handle different representations.

In REST, access to resources is enabled by the *uniform interface*, a well-defined interface introducing interaction transparency. Another constraint important in the context of resource access is the concept of *uniquely identifiable resources*. The combination of these two concepts, the uniform interface and unique identifiers, enable caching, which is one of the main strengths of REST and one of the main reasons for the scalability of REST compliant system.

In order to fulfill the constraints of the uniform interface and unique identifiers, a REST service composition has to support both, the uniform interface (requirement 2) as well as addressing resources by their unique identifier (requirement 3).

As an additional step towards loose coupling between client and server, the REST architectural style demands resource representations to follow the *Hypertext as the Engine of Application State (HATEOAS)* constraint. This constraint implies that the representation of a resource contains metadata describing possible interactions with the resource as well as its relations to other resources. These relations are typically provided as links. Following the HATEOAS constraint, all interaction with a resource is driven by the resource representation and not by any other “out of bound” mechanisms.

There can be three requirements inferred from the HATEOAS constraint. As interaction parameters (e.g. query parameters) and also the set of available resources are determined at runtime, a REST service composition has to enable dynamic partners (requirement 4) and dynamic parameters (requirement 5). In addition, when accessing a resource, not only its data but also the associated metadata has to be accessible (requirement 6).

The *state transfer* constraint denotes, that a REST compliant server is not allowed to keep any application state. All interaction related data, i.e. the state of an interaction, has to be contained in each message sent to a server. As a consequence, the state of an interaction (or session) has to be managed by the client rather than the server. This constraint also contributes to the scalability of REST systems as it avoids server affinity and eases horizontal scaling.

The state transfer constraint provides the last requirement for REST service composition. A REST service composition has to provide means to manage the state of an interaction over several single interactions (requirement 7).

The set of constraints introduced and discussed so far is not complete. Another aspect of REST is a *layered client-server* based architecture. However, this constraint does not add any requirements to service composition. The *code on demand* constraint is defined as optional and not considered here.

III. THE BPEL COMPOSITION LANGUAGE

Instead of defining a new composition language for REST services from scratch, we aim at reusing an existing, mature and standardized composition language. This approach offers several advantages. We can keep and furthermore use existing language features and we can benefit from existing tooling, for example for modeling, execution, monitoring or auditing. To summarize, the extension of an existing language rather than defining a new one promises to “obtain much with little effort”.

A. The BPEL composition language

The *Web Services Business Process Execution Language (BPEL)* is the dominating service composition language in the field of web services. It has been standardized by the OASIS consortium[1] and is widely adopted in research as well in industry [15] [16]. The BPEL standard defines XML based syntax as well as the corresponding execution semantics for process based web service composition.

BPEL provides means to define complex interactions with multiple web services, i.e. to call web service methods as well as to receive web service calls. A BPEL process in turn is offered as a web service, this is referred to as a *recursive composition model*. BPEL combines two different process modeling paradigms. On the one hand, it supports a block oriented modeling approach combining process activities and control flow structures in an interlaced way. On the other hand, it also allows modeling a process following a graph based approach, i.e. to connect process activities with control flow connectors.

[1] https://www.oasis-open.org/

Data handling inside a process is supported by *variables* and the *assign* activity. The type system of BPEL is by default based on XML schema and variable definitions consequently refer to XML schema types. The assign activity provides one or more data manipulation instructions that prescribe data transfer between variables. Accessing and identifying the relevant parts of variables is realized using XPath expressions.

BPEL offers several communication activity types for exchanging messages. The *receive* and the *pick* activity represent the reception of an incoming message by a BPEL process. A receive activity accepts exactly one message type whereas the pick activity can handle multiple message types. A pick activity can be also described as a *polymorphic receive*. For the purpose of sending a message, BPEL offers the *invoke* activity and the *reply* activity. The reply activity is used to send a message in response to a request message received by a previous receive activity. The *invoke* activity models calling a web service, i.e. sending a request message and afterwards receiving a response message.

In addition to the communication activities introduced before, BPEL offers further communication related modeling constructs. *MessageExchanges* connect a receive activity of a process with the corresponding reply activity; they manage request and response message pairs inside a process. *PartnerLinks* describe the interaction between a process and a web service in terms of which interfaces they provide each other. This is especially important in asynchronous interaction scenarios where both interacting partners call each other. In such scenarios both partners, the process as well as the web service, offer an interface the other partner depends on. *CorrelationSets* are used to unambiguously assign incoming messages to a process instance. A correlationSet basically prescribes which part of an incoming message has to be matched with which part of a variable of a process instance.

Besides communication activities, BPEL offers several so called *structured activities* describing the control flow structure of a process. Each of these activities contains one or more child activities and prescribes in which order they have to be executed. The *sequence* activity models the sequential processing of its child activities. The *if* activity is used to describe conditional behavior, the execution of its child activities is bound to conditions. The *while* and the *repeatUntil* activities describe repetitive execution, i.e. loops. Similarly to the while and the repeatUntil activities, the *forEach* activity can be used to iteratively execute its child activities. Additionally, it also supports the parallel execution of its child activities. A common use case for the forEach activity is the processing of a set of data, e.g. an array of data. Whereas the structured activities discussed so far follow the block oriented modeling paradigm, the *flow* activity introduces a graph based modeling approach. A flow activity contains a set of child activities together with a set of *links*. Each link represents conditional control flow between two activities, it defines one source activity, one target activity and an optional transition condition. A flow activity is typically used to model parallel control flow structures.

BPEL provides further language elements targeting error handling and robustness. *Scopes* in general define a common context for the elements they contain. Variables defined inside a scope are bound to this scope, in terms of visibility as well as existence. Scopes can also have so called *handlers* attached. *Event handlers* are used to handle events, i.e. they model control flow that is outside of the regular control flow of the process. *Fault handlers* prescribe how to react to faults that may occur inside a scope. Whenever a fault occurs, it is forwarded to the surrounding scope. When a matching fault handler is defined, it is executed; otherwise, the fault is propagated to the next surrounding scope. Through the definition of *compensation handlers* BPEL supports long running business transactions [17]. Before the fault handler of a scope is executed, the compensation handlers of all contained activities are executed in their reverse execution order. This allows performing domain specific undo or cleanup steps in case of a fault.

BPEL is thoroughly designed to be extensible. A BPEL XML document can be freely extended by any XML element outside of the BPEL namespace. Such extensions can be declared as optional or as mandatory, i.e. it can be declared if a BPEL engine can ignore such extension elements or if it has to support them. Besides this very general extensibility, there are two more specific extension capabilities explicitly included in the BPEL language. The assign activity can contain so called *extensionAssignOperation* elements providing additional data handling functionality. In addition, BPEL also defines an *extensionActivity*. This activity acts as a placeholder for custom activity types. If declared as optional, a BPEL engine may ignore these activities, otherwise it has to support them, i.e. it has to be able to execute them.

B. Existing Approaches for REST Composition with BPEL

Although BPEL is tightly coupled to WSDL based web services, typically realized as SOAP services, it is in parts possible to use BPEL to interact with REST services. In the following we will give a short overview about the most relevant approaches and show, why they do not provide an appropriate solution for REST service composition.

BPEL relies on the *Web Services Description Language (WSDL) 1.1* as description language for service interfaces [18]. WSDL 1.1 allows defining a HTTP binding, i.e. to map the operations of a web service to HTTP calls. Such a binding is static; the addresses of all resources have to be known in advance. There is only very limited support for content negotiation, and the mapping of a resource providing several methods to a corresponding WSDL descriptions results in an excessive set of bindings and ports to be defined.

The Apache Orchestration Director Engine (ODE)[2], an open source BPEL engine, defines a custom extension for HTTP binding in WSDL 1.1. This extension supports enhanced manipulation of resource URIs at runtime, but the host has to be furthermore known in advance. The mapping of HTTP interactions to WSDL operations results in cleaner WSDL descriptions. There is however no improvement regarding the limited support for content negotiation.

The successor of WSDL 1.1, the WSDL 2.0 standard [19], provides an enhanced HTTP binding that in parts overlaps with the WSDL 1.1 extensions defined by the ODE project. In addition, it provides basic support for content negotiation through the definition of input and output serialization. The main drawback in context of BPEL is that the BPEL language is closely bound to WSDL 1.1 and does not support WSDL 2.0 at all.

A commonality of all approaches discussed so far is, that all aspects related to REST are not visible in the composition language itself. The enabling of REST service composition is realized as a kind of deployment configuration or service binding. In contrast to that, our approach for REST service composition with BPEL explicitly defines REST interaction capabilities as part of the composition language.

IV. A BPEL Extension for REST Service Composition

In this section we present an extension to the BPEL composition language that enables the combined composition of WSDL based web services and HTTP based REST services. A main feature of the extension is standard compatibility, i.e. the resulting BPEL processes are still standard compliant BPEL processes. To achieve this goal, we build on extension capabilities already defined in the BPEL language, namely extension activities. In the following, we will introduce the meta model of the proposed extension activities. After that, we will show how these extensions fulfill the general requirements to RESTful compositions previously identified in section II.

A. The Meta Model

In order to enable the composition of REST services, we extend the BPEL composition language with a set of extension activities. For each of the main HTTP methods we define a corresponding REST extension activity. The meta model of these extension activities is shown in Fig. 1 as UML class diagram. Classes already defined by BPEL are colored grey; everything else is part of the newly defined meta model for REST service composition.

Each REST extension activity inherits from the *ExtensionActivity* class. This class contains the standard attributes and elements the BPEL standard defines for each activity. For reasons of comprehensibility, the meta model depicted in Fig. 1 only shows the optional *name* attribute.

The class *RESTActivity* is the base class for all REST extension activities; it contains attributes and elements common to each REST extension activity. The *host* and *path* attributes together identify the resource to interact with. These attributes can be provided as literals, i.e. the host or the relative path of a URI is predefined by an activity. Instead of predefining the URIs of resources, the more common way of interacting with REST services is driven by links (as defined by the HATEOAS constraint). When a representation of a resource is accessed, this representation can contain links identifying related resources. Theses links are then used to access other resources, i.e. the URI of a resource is in general

[2] http://ode.apache.org/

only known at runtime. In the context of REST service composition that means that the target URI of a REST extension activity is in general not known in advance. It typically depends on another REST service interaction performed some time before in the same composition. Therefore, as shown in the meta model in Fig. 1, each REST extension activity can also refer to BPEL variables containing the URI of the resource to interact with.

```
<variable name="host" type="string" />
<variable name="path" type="string" />
…
<GET host="localhost:8080"
     path="/api_root/">
  …
</GET>
…
<!-- read response, fetch next link -->
<!-- write link data to variables -->
…
<GET host="$host$" path="$path$">
  …
</GET>
```

Listing 1: Resource identification example

A simple example of both approaches for resource identification is given in Listing 1. At first, two string variables are defined. The first GET activity accesses a resource with a predefined URI, the host as well as the relative path are given as literals. After the first GET activity is finished, the retrieved data can be read and, depending on the domain logic, a suitable link contained in the data can be selected. In our example, we assume that the selected link data is then written to the previously defined variables *host* and *path*. This variables are then be used by the second GET activity to identify its target resource. As shown in Listing 1, the second GET activity does not contain any literals but instead references BPEL variables, indicated by the surrounding '$' characters.

Another commonality between all REST activities is defined by the *Context* class. A context defines data and configuration that applies to a set of requests, typically as part of an interaction with multiple resources of the same REST service. Typical parameters of such interactions are abstracted from the underlying HTTP header fields and modeled as context attributes (closeConnection, username, password, cacheControl). In addition to this, a context also allows to define values for arbitrary header fields. Another important aspect of a context is that it represents the state of an interaction; it acts as a container for state data like for example HTTP cookies. REST extension activities that refer to the same context share the same interaction state. As shown in the meta model in Fig. 1, the context is modeled as BPEL variable. The structure of such a context variable is well defined by a given XML schema document. Consequently, it can be created and manipulated using standard BPEL constructs.

A simple example for the usage of a context is given in Listing 2. At first, a variable named *ctx* of the predefined type *rest:context* is declared and initialized. It defines username and password to be used if an interaction requires authentication and it also defines that the underlying HTTP connection should be kept open. The GET activity then references the defined context using the *ref* attribute. The following POST activity references the same context, i.e. it is executed using the same configuration and also using cookie data possibly written during the first GET activity. In addition to referencing the context *ctx*, the second activity extends the context by defining the value *ODE-v2* for the HTTP header field *User-Agent*. It is in general possible to extend or adapt a referenced context on a per request base. A context attribute defined inside an activity always supersedes the same attribute defined by the referenced context variable.

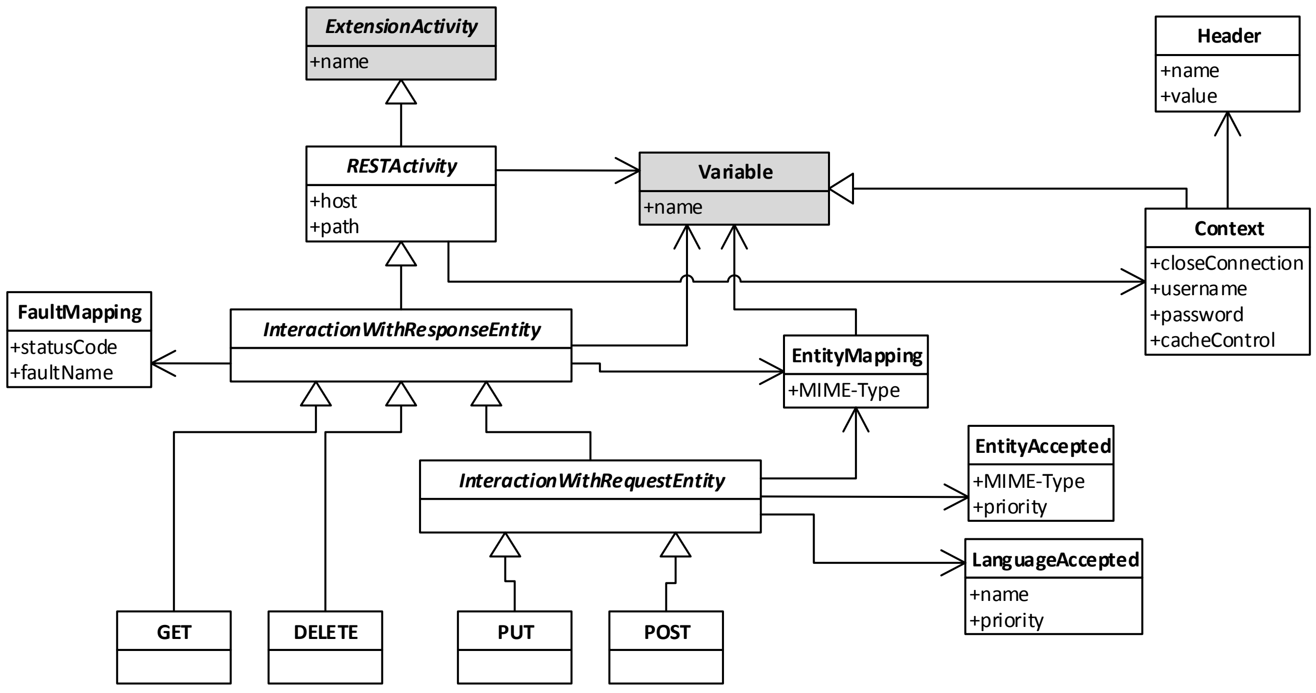


Fig. 1. A Meta Model for REST Extension Activities

```
<variable name="ctx" type="rest:context">
  <literal>
    <context>
      <closeConnection>false</closeConnection>
      <username>JohnDoe</username>
      <password>1337</password>
    </context>
  </literal>
</variable>
…
<GET host="…" path="…">
  <context ref="ctx" />
  …
</GET>
…
<POST host="…" path="…">
  <context ref="ctx">
    <header name="User-Agent">ODE-v2</header>
  </context>
  …
</POST>
```

Listing 2: Context usage example

All REST extension activities shown in Fig. 1 are able to return a response entity. To enable further processing of the result of request, the response entity is saved to a BPEL variable. According to the concept of resource representations and content negotiation, a request can possibly return different representations of the same resource. Consequently, for each possible representation a separate target variable of the corresponding data type is needed. In our meta model, this is supported by the *EntityMapping* class. A REST extension activity can refer multiple of those mappings. Each mapping defines the MIME type it applies to and also references a BPEL variable. The referenced variable has to be of a data type compatible with the given MIME type. In addition to the response entity, each request also returns HTTP header fields. These header fields can optionally be written to a BPEL variable also defined by the EntityMapping class. This enables to arbitrarily process any header fields if needed. If a request to a resource results in an error, in HTTP this is signaled by a corresponding status code. To be able to handle such faults in BPEL, for each request a *FaultMapping* can be defined. This mapping defines which BPEL fault to throw for which status code. These BPEL faults can then be handled using standard BPEL fault handlers.

A simple example for response handling is shown in Listing 3. At first, two variables are defined. The variable *pic* is of the type *base64Binary*, it is supposed to contain a picture in base64 encoding. The variable *desc* is of the type *string* and supposed to contain the corresponding description text of the picture stored in the variable *pic*. The GET activity then contains two different entity mappings. The first mapping defines that when the request returns an entity with the MIME type *application/octet-stream*, the entity data has to be stored in the variable *pic*. However, when the request returns an entity with the MIME type *text/plain*, the entity data has to be stored in the variable *desc*.

```
<variable name="pic" type="base64Binary" />
<variable name="desc" type="string" />

<GET host="…" path="…">
  <response>
    <acceptEntityMapping
       type="application/octet-stream"
       variable="picData" />
    <acceptEntityMapping
       type="text/plain"
       variable="picName" />
  </response>
  …
</GET>
```

Listing 3: Response mapping example

The concept of resource representations and content negotiation does not only affect the handling of responses but also the way requests are handled. In contrast to response entity handling discussed so far, request entity handling only applies to some activities, namely PUT and POST. These are the only activities that may contain a request entity; they both inherit from the *InteractionWithRequestEntity* class. Similarly to response entity mapping, the request entities may also be of different MIME types and therefore stored in different variables. In our meta model this is again modeled by the *EntityMapping* class. The only difference is that the mapping defines which variable *contains* the entity representation, in contrast to response handling, where the mapping defines where to *store* the entity representation. As already introduced in section II, for server driven content negotiation the client can define as part of a request what resource representations it can handle. The server then tries to find a match between this request and the available representations. This content negotiation is modeled by the classes *EntityAccepted* and *LanguageAccepted*. Using EntityAccepted, a client can define what representations in terms of MIME types it accepts. Similarly, LanguageAccepted allows defining which languages are acceptable for response entity representations. Both classes allow defining a *priority* value, which defines an ordering between multiple acceptable representations.

A simple example of request entity handling and content negotiation is shown in Listing 4. At first, two variables are defined. Both are meant to handle data encoded in base64, in this example some image data. The GET activity defines in the *contentNegotiation* element that it accepts data in GIF format as well as in JPEG format. The priority values indicate that GIF data is preferred. The *response* element defines corresponding entity mappings for each of the defined representations. After the GET activity has fetched the image data, it is processed and then again accessed by the following POST activity. In our example we assume that the GET activity fetched the GIF representation. Consequently, the entity mapping inside the POST activity defines that the request entity is of the MIME type *image/gif* and provided by the variable *gif*.

```
<variable name=″gif″ type=″base64Binary″ />
<variable name=″jpeg″ type=″base64Binary″ />

<GET host=″…″ path=″…″>
  <contentNegotiation>
    <entityAccepted type=″image/gif″
                    priority=″0.8″ />
    <entityAccepted type=″image/jpeg″
                    priority=″0.2″ />
  </contentNegotiation>
  <response>
    <acceptEntityMapping type=″image/gif″
                         variable=″gif″ />
    <acceptEntityMapping type=″image/jpeg″
                         variable=″jpeg″ />
  </response>
</GET>

<!-- process image data -->

<POST host=″…″ path=″…″>
  <requestEntityMapping type=″image/gif″
                        entity=″gif″ />
</POST>
```

Listing 4: Request mapping and content negotiation example

B. Discussion

The overarching methodology the work presented in this paper is based on is depicted in Fig. 2. In section II we introduced the REST principles relevant for REST service composition and then discussed which requirements they introduce. This part is shown in the left of Fig. 2. In the previous section, we introduced the meta model for a set of REST extension activities for BPEL. The right part of the figure shows, which parts of this meta model fulfill which requirements identified before. In the following, we will discuss these relationships in detail.

The first requirement (R1) demands support for content negotiation. In our meta model, this is realized by two different entities. At first, the meta model in general represents that entity representations are typed. This is modeled by the class *EntityMapping*, which is used by request as well as response interactions. Second, for a request, a set of accepted representations can be defined, modeled by the class *EntityAccepted* of the meta model.

The second requirement (R2) is realized by the definition of explicit activity types for each method of the uniform interface. In our meta model these are the classes *GET*, *PUT*, *POST* and *DELETE*. They represent the main methods of the uniform interface defined by the HTTP protocol.

The requirement (R3) is supported by the common attributes *host* and *path* that can be defined for each activity type. They are used to identify a resource by its URI, a unique identifier. These two attributes in addition also fulfill the requirements (R4) and (R5). The possibility to define the target URI of each request at runtime enables to dynamically select the partner, i.e. the resource to interact with. Besides that, it also allows influencing the request parameters, as they are also included in the URI.

The requirement (R6) demands the possibility to access data as well as metadata. As resource representations are mapped to BPEL variables, data is in any case accessible. Meta data can be present in two different characteristics. On the one hand, the meta data can be embedded in the resource representation. In this case, it is also available in a BPEL variable. On the other hand, the meta data can be provided as HTTP header fields. Again, in our meta model header fields and their content are mapped to BPEL variables, they are therefore also accessible.

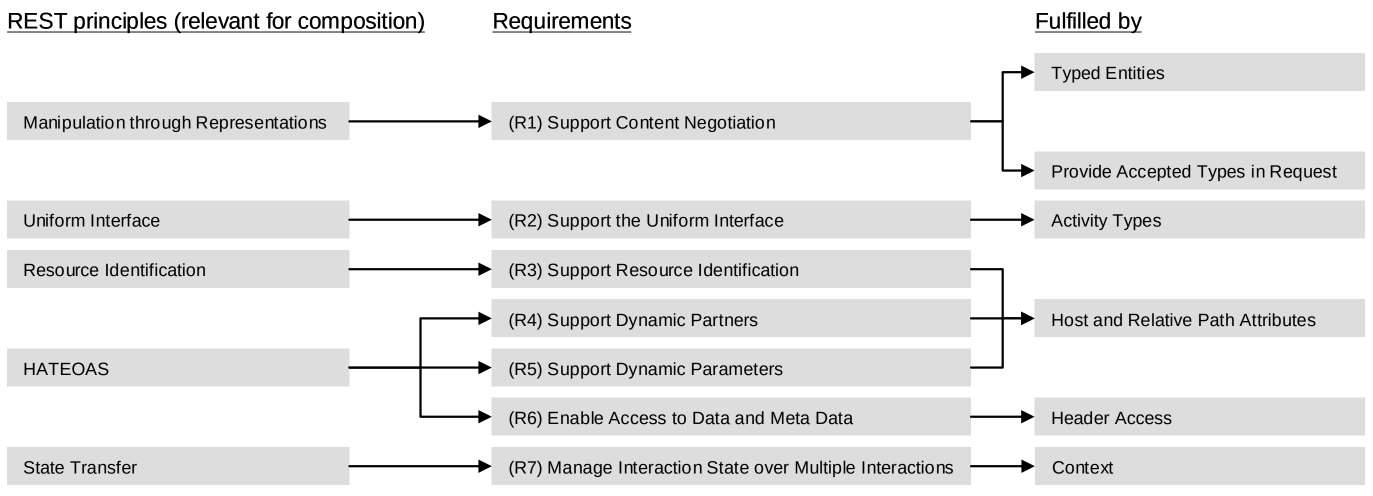
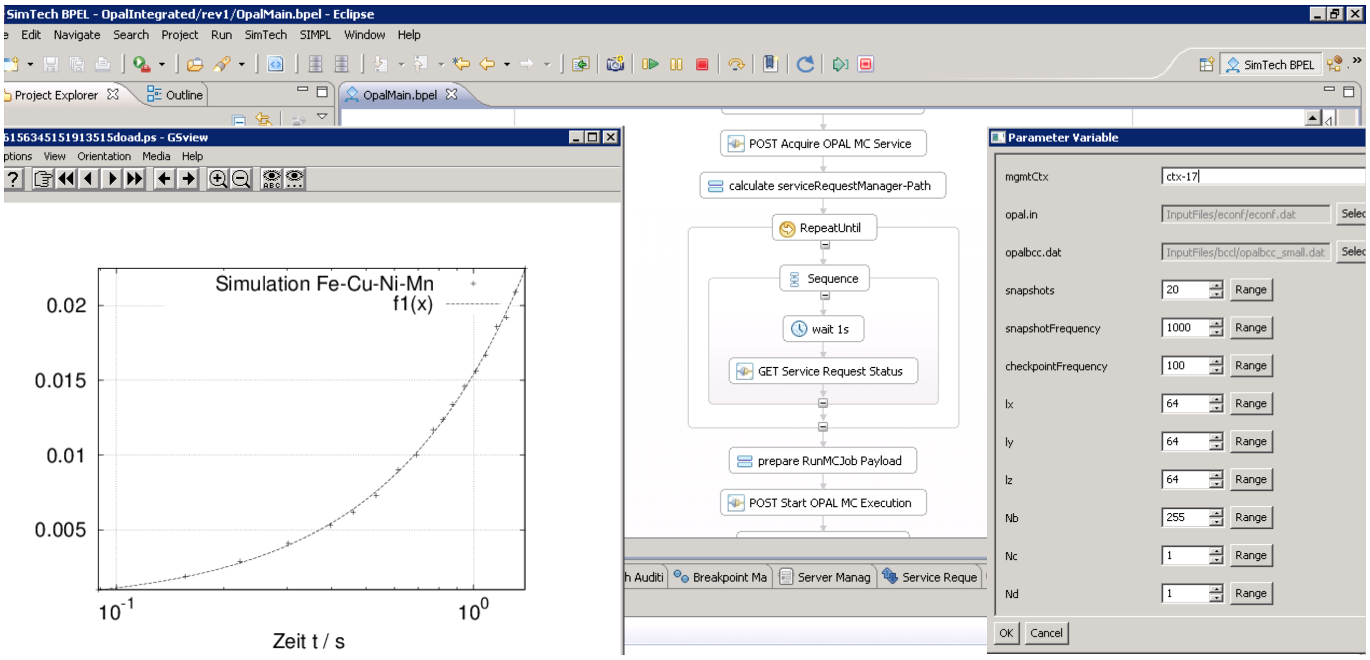

REST principles (relevant for composition)
Requirements
Fulfilled by
Manipulation through Representations
Uniform Interface
Resource Identification
HATEOAS
State Transfer
(R1) Support Content Negotiation
(R2) Support the Uniform Interface
(R3) Support Resource Identification
(R4) Support Dynamic Partners
(R5) Support Dynamic Parameters
(R6) Enable Access to Data and Meta Data
(R7) Manage Interaction State over Multiple Interactions
Typed Entities
Provide Accepted Types in Request
Activity Types
Host and Relative Path Attributes
Header Access
Context

Fig. 2. Methodology overview


SimTech BPEL - OpalIntegrated/rev1/OpalMain.bpel - Eclipse
Edit Navigate Search Project Run SimTech SIMPL Window Help
Project Explorer
Outline
OpalMain.bpel
SimTech BPEL
515634515191351 5doad.ps - GSview
ptions View Orientation Media Help
Simulation Fe-Cu-Ni-Mn
f1(x)
0.02
0.015
0.01
0.005
10^-1
10^0
Zeit t / s
POST Acquire OPAL MC Service
calculate serviceRequestManager-Path
RepeatUntil
Sequence
wait 1s
GET Service Request Status
prepare RunMCJob Payload
POST Start OPAL MC Execution
h Auditi
Breakpoint Ma
Server Manag
Service Reque
Parameter Variable
Range
OK
Cancel

mgmtCtx	ctx-17
opal.in	InputFiles/econf/econf.dat
opalbcc.dat	InputFiles/bccl/opalbcc_small.dat
snapshots	20
snapshotFrequency	1000
checkpointFrequency	100
lx	64
ly	64
lz	64
Nb	255
Nc	1
Nd	1

Fig. 3. SimTech SWfMS showing OPAL simulation with REST composition

The last requirement (R7), the support of state handling across multiple interactions, is mainly realized by the *Context* class. As described, it acts as a container for state data. As the context is realized as BPEL variable, a composition can contain multiple contexts and thereby manage multiple interaction states in parallel. This enables modeling compositions comprising multiple parallel interactions with different REST services, each with its own interaction state. Each activity type references exactly one context variable and therefore shares the state contained in the corresponding context.

To summarize the methodology depicted in Fig. 2, we first identified a set of general requirements for REST service composition directly from the definition of the REST constraints. Our meta model of a BPEL extension for REST service composition was then designed to fulfill exactly these requirements. Finally, we discussed for each of the requirements, by which part of the meta model it is fulfilled. Another key feature of our BPEL extension is standard compatibility. We solely used explicitly defined extension capabilities, i.e. the concept of extension activities and we realized other extensions, like for example the context, by mapping them to BPEL variables. As a result, the extended BPEL language does not lose any of its features but wins the additional feature of supporting REST service composition.

V. Realization and Validation

The BPEL extension activities presented in section IV are prototypically implemented and integrated into an open source BPEL engine, the Apache Orchestration Director Engine (ODE), Apache ODE explicitly provides an extension point to plug in execution logic for arbitrary BPEL extension activities. At deployment time, for each extension activity type contained in a BPEL process it is checked, if a corresponding implementation is available. If this check is successful, the BPEL process can be instantiated and executed. The realization of the meta model of our BPEL extension also includes XML schema definitions for context and header variables.

For validation purposes we use the *SimTech Scientific Workflow Management System (SimTech SWfMS)* for the modeling, execution and monitoring of BPEL workflows. The SimTech SWfMS is based on conventional workflow technology and has been specifically adapted to the needs of simulation workflows [11]. The workflow engine of the SimTech SWfMS is an extended version of Apache ODE, the integrated modeling and monitoring tool is based on the open source *Eclipse BPEL Designer*[3].

The use case for our validation is based on the simulation application OPAL (**O**stwald-Ripening of **P**recipitates on an **A**tomic **L**attice) [20]. OPAL implements a Kinetic Monte Carlo (KMC) simulation of the growth process of precipitates in copper and has originally been developed as a set of monolithic programs written in Fortran. In previous work, the OPAL application has been extended with a management framework and wrapped as web services [21]. As a result, the processing of OPAL based simulations has been modeled as a BPEL process executable by the SimTech SWfMS. In addition to a web service interface, the extended OPAL application is also accessible as REST service.

To validate our BPEL extension for REST service composition, we modeled the composition for an OPAL based simulation in BPEL using the extension activities presented before. The resulting OPAL process consists of four basics steps. At first, a simulation context is created, the input data for the simulation is stored in this context and the data is preprocessed using the *opalbcc* and *opalabcd* services. After that, access to the *opalmc* service, the core KMC simulation, is

[3] http://www.eclipse.org/bpel/

enquired. As soon as enough compute resources are available, access to the opalmc service is granted and the KMC simulation is started. While the simulation is running, which maybe days to weeks, at regular intervals checkpoints with intermediate results are generated. In a third step, these checkpoint data are analyzed using the *opalclus* and *opalxyzr* services. After the simulation has finished and all checkpoint data are analyzed, in the last step the *opalmedia* service is called. This service creates a simple visualization of the simulation data.

A screenshot of the graphical frontend of the SimTech SWfMS is shown in Fig. 3. In the center, an excerpt of the described OPAL simulation process is shown. In this part of the simulation, a POST request is used to enquire access to the opalmc service. This request is followed by a loop containing a GET request to regularly check the status of the submitted opalmc service request. As soon as the opalmc service is available, the loop is finished and the following POST request starts the KMC simulation. When a process is modeled, the SimTech SWfMS frontend provides a convenient way to automatically deploy and start the process. On the right side of Fig. 3, the corresponding dialog window is shown, where the user can provide input parameters needed by a process. The result of the execution of the OPAL process, a visualization of the simulation data, is shown in the left part of Fig. 3.

VI. Related Work

A first approach towards REST service composition also based on the BPEL composition language is presented in [22]. Similarly to our approach, separate activities are defined for each of the HTTP operations. However, the definition of these REST activities is only little abstracted from HTTP. Whereas the approach presented in our work supports many features by explicit and HTTP independent modeling constructs (e.g. caching, access data, content negotiation), many of these have to be realized by setting and reading low-level HTTP header fields in [22]. In addition to defining a means for composing REST services, the author also describes how to realize a REST service by using BPEL. Following the recursive composition model of BPEL, a process itself is again provided as a resource.

The *Bite* composition language proposed in [23] focuses on the domain of web mashups. It provides a lightweight process model and adopts several concepts from scripting languages, for example implicitly defined variables and data flow. In order to realize mashups of web resources, Bite provides several activity types to interact with REST services. However, as Bite is defined for the rather special use case of web mashups and in addition has several implicit functionality, the REST composition capabilities are rather limited compared to the requirements defined in our work.

In [24] the authors focus mainly on the HATEOAS aspect of REST and its implication on REST service composition. For this purpose, a composition language called *Resource Linking Language (ReLL)* is defined together with a petri net based meta model. In ReLL, the links between resources are explicitly modeled in the composition and the interaction with resources focuses on selecting and following links. While focusing on the HATEOAS aspect, the ReLL language does neglect data flow capabilities that are in contrast supported by our BPEL based approach. As the ReLL language is a research prototype, it lacks the maturity and quality of service of well-established composition languages like BPEL.

In [25] REST service composition is discussed with focus on the application domain of mashups. Similarly to our work, a set of requirements for REST composition is defined. In contrast to our methodology, no explicit connection between the REST constraints and the defined requirements is shown. The set of requirements defined in [25] and the set of requirements defined in our work have some overlapping but also some differences. Where [25] requires dynamic typing as well as content negotiation, we do only require support for content negotiation (R1) but not for dynamic typing. Instead of introducing dynamic typing to BPEL, in our solution we first perform content negotiation and then transform non-XML representations into corresponding XML representations. In addition, we define some requirements that are not, or at least not explicitly, covered in [25] (R3, R6, R7).

VII. Conclusion and Future Work

The work presented in this paper is based on the three step methodology illustrated in Fig. 2. As starting point, we performed a detailed analysis of the REST architectural style in relation to service composition. We were able to infer a set of seven basic requirements to be fulfilled by REST service composition. In a second step, we introduced a meta model for REST service composition based on the BPEL composition language. Doing so, we did not only aim at fulfilling the defined requirements, we also enabled to reuse the already available functionality of a standardized, mature and powerful composition language and to combine it with new abilities for REST service composition. In a third step, we were able to show, that all requirements were already fulfilled by the design of our meta model. The presented approach for REST service composition has been validated based on a real world use case from the eScience domain. We realized the BPEL extension as part of the SimTech SWfMS and then used it to successfully model and execute an OPAL simulation.

As part of our future work, we plan to evaluate the feasibility as well as the advantages and disadvantages of different service composition approaches. One fundamental aspect that became clear during the validation of our approach is the handling of long running operations. When using BPEL for the composition of web services, long running operations are typically realized as asynchronous operations using a callback mechanism. In contrast to that, in REST service composition based on HTTP, asynchrony is not supported. Therefore, long running operations are typically realized using a polling mechanism. This is only one example of how the different composition approaches differ, and we think it might be promising to investigate this in more detail.

Acknowledgment

This work was partially funded by the BMWi project Migrate! (01ME11055). D. Karastoyanova and K. Vukojevic-Haupt 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.

References

[1] Papazoglou, Mike P. "Service-oriented computing: Concepts, characteristics and directions." Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on. IEEE, 2003.

[2] Chen, L., Shadbolt, N. R., Goble, C., Tao, F., Cox, S. J., Puleston, C., & Smart, P. R. (2003). Towards a knowledge-based approach to semantic service composition. In The Semantic Web-ISWC 2003 (pp. 319-334). Springer Berlin Heidelberg.

[3] Benatallah, B., Dumas, M., Sheng, Q. Z., & Ngu, A. H. (2002). Declarative composition and peer-to-peer provisioning of dynamic web services. In Data Engineering, 2002. Proceedings. 18th International Conference on (pp. 297-308). IEEE.

[4] Hamadi, R., & Benatallah, B. (2003, January). A Petri net-based model for web service composition. In Proceedings of the 14th Australasian database conference-Volume 17 (pp. 191-200). Australian Computer Society, Inc..

[5] OASIS: Web Services Business Process Execution Language Version 2.0 (11 April 2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[6] Fielding, R.T.; Taylor, R.N.: *Principled design of the modern Web architecture*. In: ACM Trans. Internet Technol. 2, 2 (May 2002)

[7] Pautasso, C., Zimmermann, O., & Leymann, F. (2008, April). Restful web services vs. big'web services: making the right architectural decision. In Proceedings of the 17th international conference on World Wide Web (pp. 805-814). ACM.

[8] Mell, Peter; Grance, Timothy (2011): The NIST Definition of Cloud Computing (Draft). http://www.nist.gov/itl/cloud/.

[9] Mietzner, R., Unger, T., & Leymann, F. (2009). Cafe: A generic configurable customizable composite cloud application framework. In On the Move to Meaningful Internet Systems: OTM 2009 (pp. 357-364). Springer Berlin Heidelberg.

[10] Binz, T., Breiter, G., Leyman, F., & Spatzier, T. (2012). Portable Cloud Services Using TOSCA. IEEE Internet Computing, 16(3).

[11] Görlach, K., Sonntag, M., Karastoyanova, D., Leymann, F., & Reiter, M. (2011). Conventional workflow technology for scientific simulation. In Guide to e-Science (pp. 323-352). Springer London.

[12] Deelman, E., Gannon, D., Shields, M., & Taylor, I. (2009). Workflows and e-Science: An overview of workflow system features and capabilities. Future Generation Computer Systems, 25(5), 528-540.

[13] Pagni, M., Hau, J., & Stockinger, H. (2008, May). A multi-protocol bioinformatics web service: Use soap, take a rest or go with html. In Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on (pp. 728-734). IEEE.

[14] Fox, G. C., Guha, R., McMullen, D. F., Mustacoglu, A. F., Pierce, M. E., Topcu, A. E., & Wild, D. J. (2009). Web 2.0 for Grids and e-Science. In Grid enabled remote instrumentation (pp. 409-431). Springer US.

[15] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., & Ferguson, D. F. (2005). Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more. Prentice Hall PTR.

[16] Ouyang, C., Verbeek, E., Van Der Aalst, W. M., Breutel, S., Dumas, M., & Ter Hofstede, A. H. (2007). Formal semantics and analysis of control flow in WS-BPEL. Science of Computer Programming, 67(2), 162-198.

[17] Papazoglou, M. P. (2003). Web services and business transactions. World Wide Web, 6(1), 49-91.

[18] Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001). Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl

[19] Chinnici, R., Moreau, J. J., Ryman, A., & Weerawarana, S. (2007). Web services description language (wsdl) version 2.0 part 1: Core language. W3C recommendation, 26, 19. http://www.w3.org/TR/wsdl20/

[20] Binkele, P., & Schmauder, S. (2003). An atomistic Monte Carlo simulation of precipitation in a binary system. Zeitschrift für Metallkunde, 94(8), 858-863.

[21] Sonntag, M., Hotta, S., Karastoyanova, D., Molnar, D., & Schmauder, S. (2011). Using services and service compositions to enable the distributed execution of legacy simulation applications. In Towards a Service-Based Internet (pp. 242-253). Springer Berlin Heidelberg.

[22] Pautasso, C. (2009). RESTful Web service composition with BPEL for REST. Data & Knowledge Engineering, 68(9), 851-866.

[23] Rosenberg, F., Curbera, F., Duftler, M. J., & Khalaf, R. (2008). Composing restful services and collaborative workflows: A lightweight approach. Internet Computing, IEEE, 12(5), 24-31.

[24] Alarcon, R., Wilde, E., & Bellido, J. (2011). Hypermedia-driven RESTful service composition. In Service-Oriented Computing (pp. 111-120). Springer Berlin Heidelberg.

[25] Pautasso, C. (2009). Composing RESTful Services with JOpera. In Software Composition (pp. 142-159). Springer Berlin Heidelberg.

All links were last followed on 2014-06-18.