A Structured Marketplace for Arbitrary Services

Ralph Vigne*, Juergen Mangler**, Erich Schikuta, Stefanie Rinderle-Ma

Workflow Systems and Technology Group
Faculty of Computer Science
University of Vienna
Rathausstraße 19/9
1010 Vienna, AUSTRIA

Abstract

Creating simple marketplaces with common rules, that enable the dynamic selection and consumption of functionality, is the missing link to allow small businesses to enter the cloud, not only as consumers, but also as vendors. In this paper we present the concepts behind a hybrid service and process repository that can act as the foundation for such a marketplace, as well as a prototype that allowed us to test various real-world scenarios. The advantage of a hybrid service and process repository is, that it not only holds a flat list of services, but exposes a generic set of use cases, information how specific services can be used to implement the use cases as well as information to select services at run-time according to customers goal functions.

Keywords: Service Repository, Marketplace, UDDI, Multilevel Mircoflows

1. Introduction

The prospect of utility computing is simple: “pay and use” computing resources. The currently dominating manifestation is cloud computing. When it comes to the nature of this computing resources we roughly differentiate between IaaS - Infrastructure as a Service (virtual machines, storage), PaaS - Platform as a Service (middleware), and SaaS - Software as a Service (applications or services to be consumed by users).

Software as a Service (SaaS) allows a vendor to deliver complex functionality to a customer over the network, avoiding installation and maintenance on client computers. We differentiate between services targeted at users (exposed through an UI) and functionality consumable and reusable by other software (many SaaS applications combine both).

Of course the potential advantages for companies are manifold [1], including the abilities to (1) gain more flexibility when reacting to external influences, (2) reduce the amount of capital needed to explore new business areas, and (3) using resources more efficiently (thus saving

*Phone: +43-1-4277-395 99, Fax: +43-1-4277-9 395
**Phone: +43-1-4277-395 12, Fax: +43-1-4277-9 395
Email addresses: ralph.vigne@univie.ac.at (Ralph Vigne), juergen.mangler@univie.ac.at (Juergen Mangler), erich.schikuta@univie.ac.at (Erich Schikuta), stefanie.rinderle-ma@univie.ac.at (Stefanie Rinderle-Ma)

energy, money and reducing the carbon footprint). In this respect cloud computing is the implementation of outsourcing for computing resources [2, p. 4ff].

With increasing availability of different services, heterogeneity aspects become apparent. Making heterogeneous services (that possibly cover overlapping functionality) discoverable, and interchangeable is hard. Partial solutions range from semantic annotations (e.g. OWL-S) and service repositories to mashup and service composition techniques (e.g BPEL). However, cloud solutions with their focus on utility and simplicity show, that the complexity necessary to integrate services through these techniques should be hidden.

Computing resource vendors keep their own (although simple) interfaces. Every customer trying to integrate several cloud services, or trying to dynamically select from a set of similar services has to implement all the available interfaces and / or rely on semantic annotations provided by the vendors.

The result is a heterogeneous environment with the following generic questions to be solved, which define a working marketplace:

1. **Can differences be explored?** Can resources be selected dynamically when several vendors provide similar resources?
2. **Can independence be maintained?** Can resources be swapped dynamically without breaking applications, when several vendors provide similar resources?
3. **Can vendors be hold accountable?** Can the quality of services (or compound services) be tracked? Is it possible to provide reliable accounting for service deficiencies (cf. Service Level Agreements [3])?

This paper focuses on the first two topics, which are related to dynamic service description and selection. We envision a marketplace, a set of resources or services, where consumers can select from a variety of available services to build complex applications.

The contribution of this paper is the introduction of a *hybrid* process and service repository, that does not only list services, but describes how to interact with them, to provide for such a marketplace. The characteristics and advantages, that mark a substantial improvement in relation to current approaches, are:

Use Case Based: The Marketplace exposes a set of common use cases for a certain domain. The interaction between customers is solely based on these use cases (no specific technical details have to be considered).

Unified Invocation: No need to differentiate between SOAP, REST, ... when using services from the repository.

Unified Semantics: No need to differentiate between services from different vendors.

Passive and Scalable: Marketplaces provide no functionality besides registering vendors and reading use case related data. No single point of access is necessary, the information can be distributed to several nodes.

So instead of providing semantic information about services, that is hard to parse and hard to match, the marketplace has to provide microflows. Lambros et al. [4] define microflows as a sequence of actions taken to perform a business function provided by an operation. We extend this definition by allowing not only sequences but complex process logic (e.g. parallel splits,

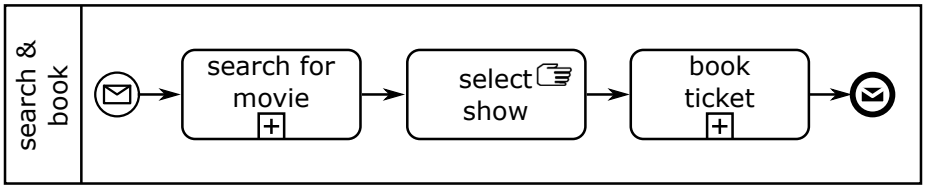


Figure 1: Cinema Use-Case

loops, ...), to allow for a greater flexibility in orchestrating the actions. In OWL-S terms our microflows would represent the service model (see Martin et al. [5]).

In order for a client to include services, it can collect microflows from the marketplace and (1) execute them locally or (2) leverage additional cloud infrastructure (a cloud process execution engine) to delegate execution and collect the result.

To prove the feasibility of our system we selected a suitable real world scenario: a trip to the movie theater. The rationale behind selecting this scenario was: (1) lots of competing movie theaters exists, (2) they all provide online information about movies, (3) they all provide slightly different ways to book movies, (4) the service interfaces to book movies are available without costs. We added over 40 real cinemas to our repository (SOAP, REST, plain websites), designed the generic use case "search and book a movie" (see Fig. 1) to be covered by the marketplace and provided the necessary information for all cinemas to comply with the use case.

Fig. 1, although simple, includes all interesting aspects, like (1) compiling a list of movies, cinemas, locations and times from a broad set of services according to user input (*search for movies*), (2) the selection of a particular service / show by a user (or user defined goal function) (*select show*), and (3) the actual booking of a ticket for a selected show, by calling a particular service (*book ticket*).

We think that marketplaces providing a high-level interface to unify heterogeneous cloud resources (including e.g. Service Level Agreement provision and negotiation [3], secure payment utility provision [6]) are the missing link in a heterogeneous cloud future.

The paper is structured as follows: in Section 2 structural and service filtering requirements, as well as heterogeneity aspects are highlighted. Section 2 concludes with requirements for client application development. In Section 3 the nature of a cloud marketplace is discussed, which leads to a detailed description of the cinema example that is used throughout the paper. In Section 4 a presentation of the concepts and algorithms involved in implementing the marketplace is given. Section 5 continues with a focus on concepts related to abstracting and adding vendor services to the marketplace. Section 6 presents related work. In Section 7 we present a run-time analysis, comparing the performance of the marketplace to OWL-S based systems. Finally, Section 8 outlines how and why secure anonymous payment and SLA negotiation play an important part in a marketplace. We conclude our work in Section 9, including a short outlook on future work.

2. Requirements and Design Goals

As stated above, for Software as a Service (SaaS) we differentiate between two kinds of applications:

- End user cloud applications are intended to replace the current "buy, install, use" cycle with a much simpler, network based "pay and use" model. For the end-users the world stays the same, it's just simpler for them to maintain software and switch between devices.

- Abstract cloud services represent a differentiation in the way how business is done in the cloud. Instead of targeting end-users directly, these services try to deliver benefit or information that is to be leveraged by end-user cloud applications. They represent the migration path from a cloud world with flat consumer - vendor relations to a differentiated market based cloud with complex customer - vendor - vendor relations.

As stated by Schubert et al. [1, p. 8pp] cloud systems can be categorized by different aspects e.g. stakeholders or types. According to the proposed categories we imagine a marketplace to be operated as a public community SaaS cloud. Similar to Apple's App Store, users benefit the most when many (interchangeable) services exist inside a service domain. Potentially this leads to very high-quality results for the user-requirements. Consumers and vendors benefit because leveraging these replaceable services reduces their dependence on external influences. For vendors the reduced amount of money needed to provide a service (by relying on other basic services), can accelerate the creation and adoption of service based clouds. Community clouds overlap with grid technologies, specially eBusiness grids, which aim for similar goals.

Marketplaces have to allow vendors to provide services as well as to consume or refer to other vendors services, under the additional constraints: (1) services providing similar functionality can act as drop in replacements and (2) user requirements can transparently affect service use at run-time.

A marketplace for services has to deal with two different kinds of services: (1) high level services, offering powerful functionality by aggregating the interfaces and combining the execution of (2) low level, vendor specific services. The low level services have to be selected according to properties relevant to a client.

A very important goal is: the marketplace has to be passive. We do not think that the marketplace should provide own functionality, instead it has to be a repository for descriptions, which acts as a container for common rules like payment commodities (i.e. a common payment service), Service Level Agreements (SLAs) or a SLA negotiation schema.

In this section we provide a detailed discussion of these requirements and goals.

2.1. Heterogeneity Aspects: Parameters and Microflows

Services covering similar use cases, may have different APIs and use different service technologies (SOAP, REST, ...) to expose their functionality. Therefore a unified way to describe how to interact with services has to be provided. Services consist of two parts: (1) **interfaces** containing in- and output parameters (syntax) and (2) **microflows** describing ways to interact with them (semantics).

Vendors of a similar low level services need similar input data. The parameters of their interfaces may have different names, types and formats but in general they are very similar. High level interfaces aggregating interfaces of low level services provide always subsets of the functionality of actual low level services. The marketplace can thus enforce design time checks if a market participant (vendor) sticks to the marketplace rules.

Even if some parameters of a low level service interface are semantically equal they may have different formats. It is also possible that a vendor provides the defined functionality using more then one service calls and therefore must assign parameter subsets to different service calls. An additional case is, that a vendor service needs (1) additional or (2) less data than provided by the high level service. Case (2) highlights, that although the semantics provided by the high level service is a subset of the semantics provided by the low level services, the parameters themselves

have to be a superset. Case (1) is only possible if it is possible to define sane, immutable defaults for low level services.

Using the full power of a process description language (like BPEL) to describe how each low level service must be invoked to provide a functionality described by a high level service allows a repository not only to function like UDDI yellow pages [7], but also to provide a foundation for interaction with services.

Heterogeneity is not a technical obstacle but a strength of a marketplace.

2.2. Two-Level Structure

The high level (**class level**) provides well defined functionality to a marketplace customer (e.g. in terms of a farmers market: a common metric system, a set of terms to describe and thus compare fruits and a common way to pay for fruits etc.). Therefore this level defines an **abstract service description** (see also Fig. 3(b) for details). This description has to include

- **class level operations**. In Fig. 2 each of the three expanded pools represents one class level operation. The calls to services may target (1) other class level operations (e.g. *call class search* in Fig. 2), (2) operations provided by services of vendors (e.g. *perform search* in Fig. 2) or (3) external resources (e.g. *select show* in Fig. 2). For describing the class level operations (and their three targets) a process description language (e.g. BPEL) is suitable.
- a **schema for properties**. It defines what data each service has to provide. This data could be used for an appropriate selection of a particular service. This includes static information like location, that is immutable and can function as a base for structuring the marketplace (arranging services into groups and subgroups).
- a **schema for service description**. By defining one schema for all services within an application domain we encourage service vendors to use this schema. In order to add their service to the repository they have to provide a working translation to the service description, as described below.

On the low level (**instance level**) the heterogeneous resources / services are annotated and assigned to the generic high level functionalities in order to make them exchangeable. Our survey showed that each low level service, even if it provides similar functionality, has its own way to realize it. In order to express the invocation of a service in terms of the abstract description we can again leverage a process description language.

Doing so allows a vendor to keep its own interface and supports the inherent heterogeneity demands. In Fig. 4 we show three example instance level descriptions for the *search* operation provided directly by each vendor. Each of these three processes represents one kind of service we have elaborated in our survey.

2.3. Client Perspective

While UDDI / OWL-S [5] concentrates on allowing computers to discover and use heterogeneous services at run-time, the marketplace concentrates on providing use cases, which are selected at design-time (by humans). In order for a client to use the cases defined in an *abstract service description* it has to implement class level operations. As the class level operations are represented by microflows, these microflows have to be executed by a suitable process execution engine (depending on the type of process execution language exposed by the marketplace).

During the execution of a class level microflow, instance level microflows are included dynamically, i.e. by spawning sub-processes.

3. A Sample Marketplace

To exemplify our approach we selected a particular domain: online cinema services. This field seemed particularly suited because a wide array of independent resources related to searching and booking movies exists. We conducted an extensive survey for over 40 cinemas to find out and classify:

- Which kind of functionality is provided?
- What are the technical properties of the provided functionality?

This survey is the basis for the example that is used throughout this paper. It also provides the base data used in our implementation.

The cinema domain yields a number of subgroups for a closer specification of the offered services, like Arthouse, Multiplex, Drive-in. The common (and simple) functionality for cinemas is: **search for a given movie** and **book tickets for a particular show**. All cinemas use different techniques and interfaces to represent their services. Some Cinemas provide services using **SOAP / WSDL**, others offer **REST** interfaces. Some cinemas even provide **static websites** and rely on emails for the booking process.

Finally we collected static information about cinemas: service vendor name and email, address, geo-location, ...

After analyzing this data we came up with the use case shown in Fig. 2, to leverage the interchangeable services:

Search: Customer use this operation to search through the shows of a vendor for a particular movie and date. It returns an XML list with detailed information about each matching show.

Book: Customers use this operation to book tickets for a given show. It returns the reservation ID needed to pick up the tickets at the counter.

Search & Book: This operation searches through the shows of all vendors within a given resource for a particular show on a particular date. The return of each vendor is aggregated into a list and sent to the resource provided by the customer. We assume that in our example this resources allows the customer to select a show. At the end the system books the tickets for the selected show.

4. Implementation

To allow easy access to our marketplace we decided to realize it as a simple RESTful application (see Fielding [8]). The marketplace allows vendors to register and maintain their services and domain experts (the marketplace operators) to create and maintain application domains.

Fig. 3(a) shows the structure of the RESTful marketplace. It is organized in a hierarchical manner: the class level is represented at the top, with the properties spanning a resource tree in the middle, and the actual instance level services in the branches. As the marketplace is only

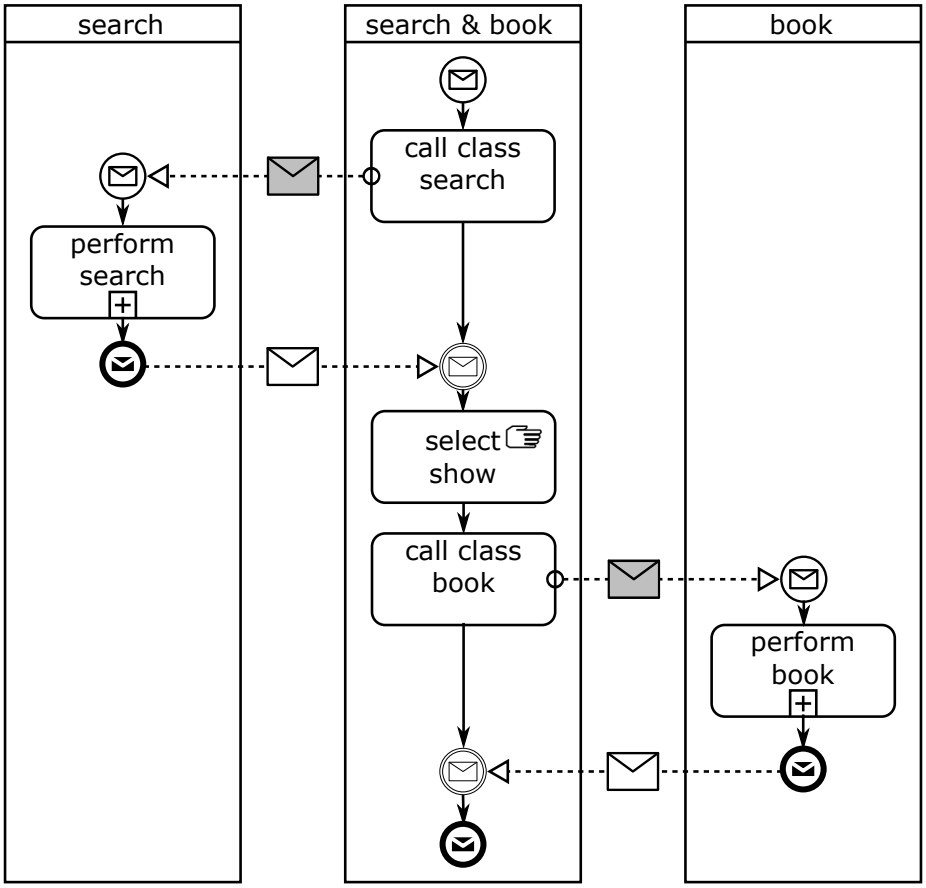
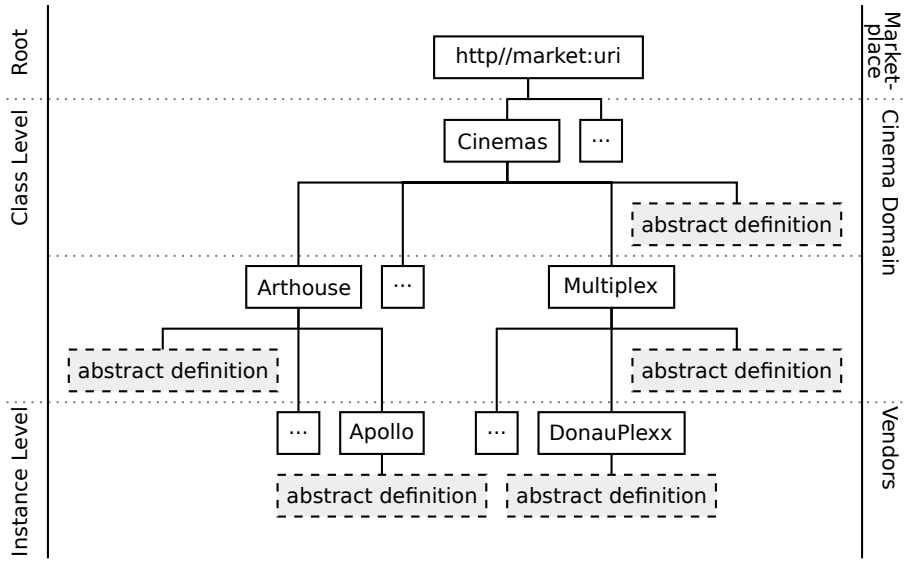
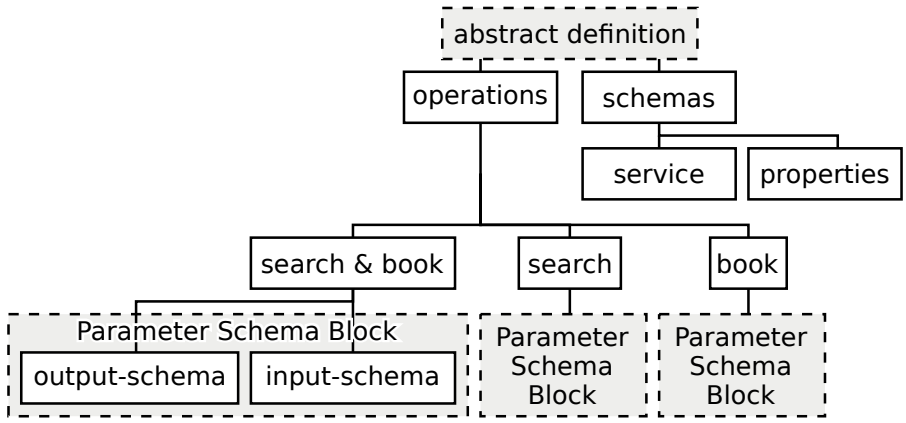


Figure 2: Inter-Dependencies of Class Level Operations



(a) Yellow Information



(b) Green Information

Figure 3: Structure of the Marketplace

passive (no functionality apart from reading stored information) it is easy to split up this tree of static documents, distribute it to several nodes and make it accessible through http-proxy load-balancing [9], thus providing scalability and the ability to merge independent marketplaces.

4.1. Schemas

From every node in the RESTful marketplace application it is possible to reach the abstract service definition (see Fig. 3(b)) which represents the use cases / green information in UDDI terms. The schemas included in the abstract definition have the following functions:

- Allow new service vendors to perform a static check of the information (and the format) they provide to the marketplace. In reverse the marketplace can perform these checks on registrations (or changes).
- Allow customers to check for changes in the use cases.

There are two separately accessible schemas available, both expressed in XML RNG [10] schema:

4.1.1. Service Schema

The service schema allows the marketplace to perform *structural* checks on services provided by service vendors. It has the purpose to ensure that all services provide the necessary operations and properties. It can also act as a central point for providing:

- documentation for abstract service descriptions,
- payment rules common to all services in this domain of the market place, and
- other global SLAs common to all services in this particular domain of the marketplace.

From this class level schema, an instance level stub can be generated automatically. The service schema consists of three parts:

- The properties schema as defined in the next sub-section. This schema is duplicated here to allow for checking all information a vendor provides to the marketplace in one step.
- A list of instance level operations generated by parsing the abstract service description (level 1) and extracting a list of instance level service calls.
- The controlflow schema depending on the used description language (e.g. BPEL schema).

4.1.2. Properties Schema

Properties (as described in Section 2) provide a way to query services within a class during the execution of the process. This allows for service filtering.

The purpose of this schema is to allow for the definition of information used to structure the repository and thus aiding the service filtering. All properties have to be provided by the service vendors.

For the cinema domain an example properties schema looks as follows:

List. 1: Properties Schema

```
<rng:grammar xmlns:datatypeLibrary="..." xmlns:rng="...">
  <rng:start>
    <properties xmlns="..." xmlns:d="...">
      <rng:element name="address">
        <d:caption xml:lang="en">Address</d:caption>
        <d:caption xml:lang="de">Adresse</d:caption>
        <rng:element name="street">
          <d:caption xml:lang="en">Street</d:caption>
          <d:caption xml:lang="de">Strasse</d:caption>
          <rng:data type="string"/>
        </rng:element>
        ...
      </rng:element>
    </properties>
  </rng:start>
</rng:grammar>
```

The example includes a property *address* with some nested fields like *street*. The **captions** can be used by clients to generate input UIs.

4.2. Class Level Operations

A list of operations and the according **class level microflows** are provided on each level in the tree to increase flexibility of the marketplace. When the marketplace is distributed over different nodes, and the proxy system fails it is still possible to find the according class level information for services stored in branches. Secondly, as the marketplace root is not necessarily correlating with the server root (e.g. the marketplace may be located at *http://test.org/something/...*), access to class level information is much simpler when it is possible to have it available at every branch of the tree (as it is the same for every branch in a domain anyways).

To enable marketplace customers to use them, a **list of defined operations** is made available in the following form:

List. 2: List of Operations

```
<operations xmlns="http://rescue.org/ns/domain/0.2">
  <operation name="search" short="Search for shows"/>
  <operation name="search_and_book" short="Search for shows, select show and book tickets
      "/>
  <operation name="book" short="Book tickets for show"/>
</operations>
```

The list of operations is generated by parsing the class level information (extracting calls from the process definition) and thus has not to be defined and maintained explicitly.

The name (ID) is used in (other) class level microflows to refer to an operation. The name is also used when querying information about the operation through the RESTful marketplace interface: *http://test.org/something/Cinemas/operations/search*.

4.2.1. Description

A single **class level operation** is defined by a class level microflow describing how other operations and resources are used (for an example see Fig. 2). Expressing the microflow in a process description language (like BPEL), we also support the definition of data elements to store local information or state while executing operations. Class level operations can also use and combine other class level operations and thus provide new, more complex functionality. This technique enables vendors to provide meta-operations thus generating additional business value through providing new / simple high-level API (much in line with the core cloud principles).

To illustrate how the operations within an application domain are interrelated we give a short explanation of the three different types of calls that are allowed to be made through class level microflows:

Calling other class level operations allows to build more complex class level operations by reusing already defined ones and combining them with other calls within a new operation. Using this type of calls allows us to realize the method *search* & *book* in our example.

Calling instance level operations invokes functionality of a vendor at instance level. Each instance level operation itself represents a microflow describing exactly how to interact with a particular vendor service. In our example these are the operations *search* and *book*.

Calling external resources targets the invocation of any resource providing a RESTful or SOAP interface. We use this type to include functionality not intended to be provided by the marketplace, but being part of the common rules for the marketplace, like the selection algorithm for services (like *select show* in our example). Of course this may include referring to human tasks (i.e. UIs waiting for user input). In fact *select show* (which refers to selecting the actual cinema / time) is a human task.

4.2.2. Parameters Schema Block

Within class level operations, a call to an instance level operation depends on a set of input and output parameters.

The schema is generated by finding all parameters and endpoints defined within a microflow. It is necessary to ignore all intermediate parameters, that are used to transfer data between two subsequent calls.

List. 3: Algorithm for Parameter Generation

```
1  # Collect input data
2  ADD all inputs to inputs array
3
4  # Collect output data
5  ADD all outputs to outputs array
6
7  IF input schema is requested
8    FOR EACH input FROM inputs array
9      DELETE input IF input is assigned by an output
10   END
11
12   # Collect additional endpoint placeholders
13   COLLECT all placeholders and add to inputs array
14   RETURN inputs
15 ELSE # output schema is requested
16   FOR EACH output FROM outputs array
17     DELETE output IF output is assigned by an input
18   END
19   RETURN outputs
20 END
```

A class level microflow can include endpoint placeholders. They allow customers to use their own services during the execution, like e.g. the selection of specific resources according to a users goal function. An example for this is *select show*. As endpoint placeholders have to be provided by the customer, they are added to the list of necessary inputs.

Again we use XML RNG schema standard to describe these interfaces (including the definition of nested elements and complex data types):

List. 4: Extracted Input Information

```
1 <rng:grammar xmlns:datatypeLibrary="..." xmlns:rng="...">
```

```
2   <rng:start>
3     <rng:element name="additional_endpoints">
4       <rng:element name="selector_service">
5         <d:caption xmlns:d="...">Used at call(s): perform_select</d:caption>
6       </rng:element>
7     </rng:element>
8     <rng:element name="input-message">
9       <rng:element xmlns:d="..." name="title">
10        <d:caption xml:lang="en">Movietitle </d:caption>
11        <d:caption xml:lang="de">Filmtitel </d:caption>
12        <rng:data type="string"/>
13      </rng:element>
14      <rng:element xmlns:d="..." name="date">
15        <d:caption xml:lang="en">Date </d:caption>
16        <d:caption xml:lang="de">Datum</d:caption>
17        <rng:data type="date"/>
18      </rng:element>
19    </rng:element>
20  </rng:start>
21 </rng:grammar>
```

As shown in List. 4, each parameter has a data type and one or more caption elements. The multi-lingual captions can be used to generate UIs either for data input or for presenting results to customers. The data schema can be used to validate data sent to and received from the class level operations.

5. The Vendors Perspective

As explained above the instance level is about real services. This includes a set of properties for a specific service as well as microflows that describe how to invoke a particular service to implement instance level operations.

All instance level information is validated against the schema provided at class level (see Section 4.1.1 for details).

At the instance level calls to "external resource" are not allowed in the microflow. The instance level microflows represent atomic interactions between a particular customer and a particular vendor, after the vendor has been selected from the marketplace by matching the customers needs to the set of properties defined at the class level.

At the instance level, microflows represent atomic interactions (in database terms: transactions) with real services. As elaborated in previous work [11], we provide alternate microflows to rollback / retry these transactions: (1) compensate, (2) undo, (3) redo, (4) suspend and (5) abort.

Fig. 4 illustrates microflows of the instance level operation *search* describing normal execution transitions we used in our prototype implementation. Three kinds of vendors have been identified: vendors ... (1) only providing a simple HTML page, (2) offering a RESTful service, and (3) providing a SOAP interface. This example shows how different functionality from parameter transformation to process logic can be expressed using microflows. Because all three microflows implement the same interface and are described using the same language, they can be transparently used by customers.

As all three microflows shown in Fig. 4 are invoked through the class level operation 'search', they all share the same input format and all deliver output in the same format.

The **Input** message includes title, date and time when a user wants to watch a movie. The **Output** message includes a list of possible shows, as well as details about these shows.

For a **Static Cinema Service** the microflow is defined as follows: in the step *"transform Input"* the input date and time information has to be converted to a different format, in *"request*

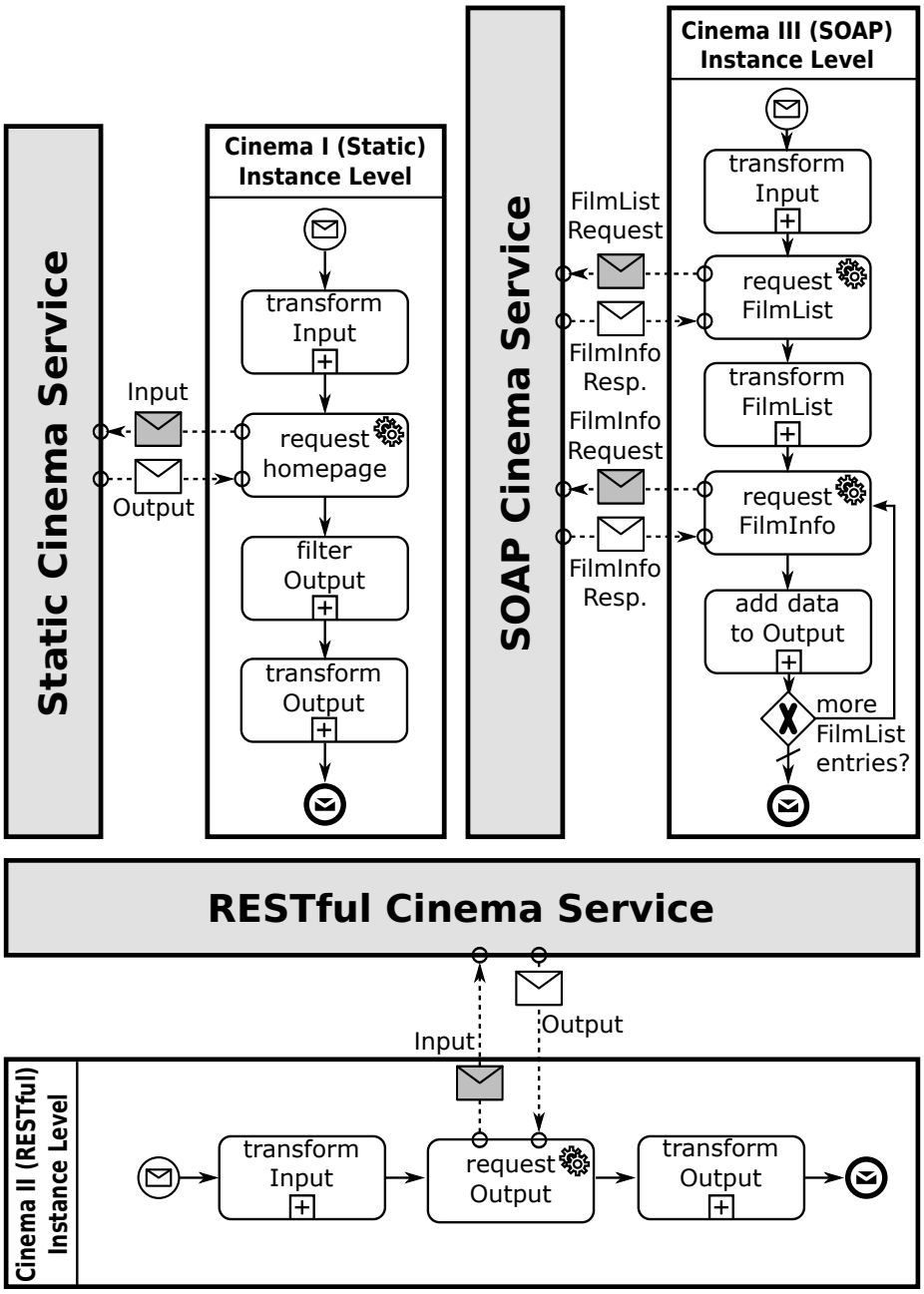


Figure 4: Three Examples of Cinemas Instance Level Microflows

homepage" the page (in XHTML format) is fetched. In the step *"filter Output"* the relevant information is extracted, and compared against the input (title, date, time, ...). The final step *"transform Output"* translates the filtered information to the necessary format expected from the (class level) operation 'search'.

For a **RESTful Cinema Service**, some steps are different. Again, the input date and time have to be converted (*"transform Input"*). Then the RESTful service is queried for results with the *Input* data (*"request Output"*). A filtering is not necessary, while the step *"transform Output"* is still available.

For a **SOAP Cinema Service** the microflows tend to be more complex. Again, the input date and time have to be converted (*"transform Input"*). After requesting a list of films constrained by the input data (*"request FilmList"*), the film id's have to be extracted (*"transform FilmList"*) and looped. For every film id, a call to the SOAP interface of the service is made (*"request FilmInfo"*), the result is then added to the output (*"add data to Output"*).

It can be observed, that the microflows for static cinema services are most resource intensive for clients and include complex transformation and extraction steps, whereas the microflows for SOAP cinema services typically contain more complex logic and produce more network traffic because of a high number of calls. RESTful cinema service normally provide a good balance between two extremes.

6. Related Work

A repository always manages metadata related to services. As pointed out in [12, 13] metadata management covers the following topics:

1. Distribution, uniformity of access.
2. Metadata should be accessible using service-oriented protocols.
3. Management of the metadata life-cycle.
4. Granular and uniform access control to metadata.

Comparing the above mentioned concepts the marketplace covers the topics 1, 2 and 4. As described in this paper, vendor maintained metadata (instance level) is available through the RESTful API. A common, reliable schema for the metadata is accessible at class level. The topic *Management of metadata life-cycle* is not covered by the marketplace. The client needs to take care about this by itself, e.g. using the techniques associated with ATOM feeds.

Bernstein et al. [14] pointed out that a repository should store information about the description of objects and the location of objects. They further stated that the most important aspect of a repository is a simple interface to interact with it. We came to the same conclusion and therefore provide a resource oriented and RESTful API. It allows customers to request ATOM feeds, listing the provided resources and further to request the schemas and microflows describing each resource.

Additionally Jha et al. [15] claimed that a system, which is intended to act as a marketplace for services, must provide two different kinds of services: (1) high level services, which offer a more powerful service and aggregate the interfaces and combine the execution of a number of (2) low level services for simpler usage. The usage of high level services is typical for cloud computing scenarios. Our service marketplace closely resembles this by providing class and instance level.

A common technique to explore services provided over the web is UDDI [7]. But as stated by Blake et al. [16] most of the resources consumed by UDDI are used for message parsing and transmissions. UDDI's weakness is, that it is just a flat list of services. Finding similar services depends on semantic annotations on the actual services. In 'A RESTful Repository To Store Services For Simple Workflows' [17] we also discuss the feasibility of UDDI in the context of service repositories.

Our approach also shares many goals with OWL-S [5]. OWL-S is used to bring semantic annotations to UDDI. Instead of simple querying for service parameters, it allows for sophisticated matchmaking. It furthermore provides a process model that describes how to use a service to achieve certain goals. It as well provides information for the actual execution. In Tab. 1 we compare the concepts of UDDI, UDDI extended with OWL-S and the marketplace.

All three concepts allow for service groups, whereas nested groups (tree structure) are only implicit in UDDI (by creatively using additional parameters). All three techniques support annotation of **attributes** to services. These attributes can be used when specifying a query. The **browse pattern** refers to returning a compact service list without all the details. The marketplace only differs from the other two solutions in the format of the returned data (ATOM). The **drill-down pattern** refers to returning detailed information for actual services. As the marketplace ensures the technical compatibility of all possible services, no services have to be excluded for technical reasons. Thus it returns potentially more services ($S_M \leqslant S_Q$, see Tab. 2 for details). The invocation pattern is supported by all three solutions. **Service Selection** holds the biggest conceptual difference between the three techniques. The marketplace focuses on "usage" or "use cases" instead of services (functionality first, services later), all services associated with a functionality fully support the use case. **Interface unification** means that different services with slightly different syntactic properties are to be used. Whereas for UDDI this has to be ensured by the client, an OWL-S enabled repository delivers information how a client may transform and invoke a service. For an example implementation of OWL-S see Srinivasan et al. [18]. With the marketplace this is not necessary, as both the transformation and invocation is contained in the microflow. By **knowledge about the item usage** we express that for the first two approaches information is returned that has to be analyzed, before calling an actual service. As the marketplace is focused on use cases, this step is not necessary.

7. Run-Time Analysis

As the marketplace is passive (data is only read, no transformations take place at server), a benchmark would only show our prototype's ability to serve static files. Groups and subgroups (e.g. cinemas, multiplex) only consist of static information. When this information is distributed to different nodes, and load-balanced by a http-proxy, a benchmark would additionally test the http-proxy's performance, and due to the simple topology it will show linear scalability. The overhead of a redirect through the http-proxy is negligible. For our prototype (the booking of cinemas) most of the time (> 99%) is spent calling the actual services (cinemas).

Therefore we decided to compare our approach to OWL-S, as OWL-S has similar goals but a very different way to achieve them. To compare the performance of the two approaches we compare how resource intensive typical steps are.

In Tab. 2 we show six steps that must be performed whenever a service from the registry / marketplace is used.

Service Query: Both techniques can query for services. While OWL-S searches through an ontology for semantic matches, the marketplace traverses a tree and executes a simple

Tab. 1: OWL-S vs. Marketplace

	UDDI	UDDI + OWL-S	Marketplace
Discovery			
Groups	Yes	Yes	Yes
Nested Groups	Implicit	Explicit	Explicit
Attributes	Yes	Yes	Yes
Browse Pattern	1	1	1
Drill-Down Pattern	n	n	m
Invocation Pattern	Yes	Yes	Yes
Selection			
Based on	Attributes	Semantics	Usage
Binding			
Description Language	WSDL	WSDL	abstract definition
Focused Item	Service	Service	Use-Case
Interface Unification Pattern	Client	Client/Server	Server
Knowledge About Item Usage	Client	Client	Server

Tab. 2: Run-Time Analysis OWL-S, Marketplace

	OWL-S				Marketplace			
	When is it done?	Who does the work?		Run-Time Effort	When is it done?	Who does the work?		Run-Time Effort
(1) Service Query	Run-Time	Registry/Server	med	$N * S_A * C * P_A$	Run-Time	Repository/Server	med	$1 * S_A * C * 1$
(2) Match Making	Run-Time	Client/Application	high	$S_Q * A_A^2$	Design-Time	Process- / Application Designer	none	0
(3) Request Service Model	Run-Time	Provider	low	S_M	Run-Time	Repository/Server	low	S_M
(4) Derive Input Transformation Schema	Run-Time	Generated by the Client	high	$S_M * A_A^2$	Design-Time	Service-Designer/ Provider	none	0
(5) Service Model Execution	Run-Time	Client/Application	low	$\sum_{i=0}^{S_N} E_i$	Run-Time	Client/ Workflow Engine	low	$\sum_{i=0}^{S_M} E_i$
(6) Derive Output Transformation Schema	Run-Time	Generated by the Client	high	$S_M * A_A^2$	Design-Time	Service-Designer/ Provider	none	0

Runtime Analysis Cheat Sheet

Variable	Intent	Relations
N	Classes	
S	Instances	$S_M \subseteq S_Q \subseteq S_A$
P	Instance Properties	$P_S \subseteq P_A$
C	Query Constraints	
A	Instance Interface	$A_P \subseteq A_A$
$\mathbf{E_i}$	Instance Process	$\forall i \in S_M$

parameter query (see Fig. 3(a) for details). In general the effort for parsing an ontology is $1+(N-1) = N$, where N is the number of all concepts/classes within it. The algorithm first searches the directly addressed concept and further collects all concepts related to it up to the second degree [18], which may be all included concepts (N). S_A defines the number of all services/instances stored within the ontology. Because each service (S_A) may be related to each class (N), the overall effort is $N * S_A$. In OWL-S each service provides its own conceptual ontology defining its properties (P_S) using attributes. When the query is compared with the properties of the services (P_S), the properties ontology (P_A) has to be searched for each property referenced in the constraints (C) separately. Because P_S could be equal to P_A, resulting in the relation $P_S \subseteq P_A$, the effort for property comparison per services is $C * P_A$. Therefore the overall effort in OWL-S during the query execution, where all services and its properties are checked, is up to $N * S_A * C * P_A$. Because the marketplace does not support relationships between classes at the same level, N is always 1. Furthermore, the marketplace provides strict schemas for properties definition (see Sec. 4.1.2) resulting in P equals 1 for each service, as no ontology needs to be searched. These two definitions lead to an overall effort of the marketplace of $1 * S_A * 1 * C$.

Match Making: An OWL-S based system has to compare the expected capabilities for a request, to the advertised capabilities of the services selected in step 1 (see Tab. 2 line 1) to which we further refer as S_Q given that $S_Q \subseteq S_A$. How often the ontology is searched for each service (S_Q) depends on the interface provided by the process (A_P) and the collected interfaces of all advertised services (A_A). Therefore the overall effort for matchmaking is $S_Q * A_P * A_A$. As it is given that $A_P \subseteq A_A$, the overall run-time effort is up to $S_Q * A_A^2$. This step is not necessary for the marketplace, as it is ensured that all services share a common

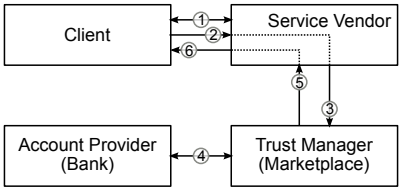


Figure 5: Secure Electronic Marketplace Transactions

schema ($A_P = A_A$), which results in an overall effort of zero.

Request Service Model: Both systems request the service model / instance level description (Section 5) for each service. We assume that the effort is equal for both systems and depends on the number of matching services from step 2 (S_M, thus there is a relation $S_M \subseteq S_Q \subseteq S_A$).

Derive Input / Output Transformation Schema: A client using an OWL-S based system has to interpret the returned information, and figure out how to transform its data to fit a service call. Some implementations e.g. Match Maker [18] generate this information during step 2 (see Tab 2 line 2) and therefore avoid additional effort without significantly increasing its effort for step 2. If not, the effort of this step is similar to the "Match Making", as again each parameter of the process (A_P) has to be transformed for each service (S_M) to fit to the service interface defined in A_A with the given relation $A_P \subseteq A_A$. Therefore the effort in OWL-S is $S_M * A_A^2$. As for the marketplace, both, the actual transformation and the service call are part of the fetched microflow. The client can supply local service endpoints that contain functionality how to create subprocesses for instance level connections, and service selection. Still the effort for the marketplace client is zero.

Service Model Execution: The service model (process in OWL-S terms) has to be orchestrated from the OWL-S client. The microflow is called for the marketplace. The effort to do so should be similar, as the concept is the same. We define this effort related to the number of instruction included in the according microflow by $\sum_{i=0}^{S_M} E_i$.

We conclude that the marketplace concept is more efficient when services have to be discovered, selected and called based on use cases. This better performance characteristics result from the fact that the marketplace does not support ontologies and not from conceptual improvements.

8. Marketplace Integration

In order for a marketplace to thrive we identified three important ingredients;

Reliable Results: as addressed in the previous sections a desired behavior of a repository of a marketplace is, that for given use-cases (class-level operations), it has always to return a result. Use-cases (see Fig. 2) are executable microflows with well defined instance level (e.g. `perform search`) or class level calls (e.g. `perform class search`):

- Calls receive a set of inputs and produce a set of outputs (as defined in the parameter schema block in Section 4.2.2).

- The user may select a set of instance level services filtered by properties as defined in Section 4.
- It is up to the user which instance level microflow to execute, they all take the same input and all produce the same output.

SLA-Mapping and Uniform SLA-Negotiation: As elaborated by Risch et al. [19], it is crucial, to ensure the all market participants are able to interpret SLA's in a common way. SLA's may be rather static properties (see Sec. 4), that can be saved in the repository, or dynamic values that need to be queried from the service each time a service is planned to use. Thus, negotiation involves the transformation of values from different services to establish a common understanding between client and services. We envision this transformation to be part of the microflows. In Fig. 2, `select show` is a manual task that presents a selection of services to the user. Or in other words: it presents a set of ranked service level agreements to the user, the steps `perform class search` and `perform search` just processes the information from different services to present a unified SLA. The microflows can thus also be used to **(a)** map different inconsistent SLA templates, and **(b)** realize a per use-case (class level) SLA negotiation. The SLA expectations are input to the use-case, measuring the performance of SLA's can again be part of the microflow.

Secure Anonymous Payment: One property of real world marketplaces is, that most small transactions can be payed in cash, or in other words, the service vendors and client engage in business transaction (including a warranty or SLA), through a trusted third party (the national bank). Both do not expose personal information such as bank accounts or names.

As elaborated by Mangler et. al [6], such a relationship is also possible for online marketplaces through gSET [20]. As depicted in Fig. 5, the marketplace functions as a `Trust Manager` between `Client` and `Service Vendor`. After ① SLA-Negotiation, the client sends a ② dual signature (created with marketplace and service vendor keys) to the service vendor. The service vendor extracts the part dedicated to him and checks the validity of the request, and then ③ forwards the request to the marketplace. The marketplace extracts and checks his part, and the ④ forwards the request to the account provider (bank). After finishing the necessary checks, a confirmation is returned to the ⑤ service vendor and to the ⑥ client. The marketplace again acts as a passive proxy. It does not store any data, but only checks messages and forwards them. Through the usage of the marketplace both parties, client and service vendor, agree on the marketplace as a trusted partner, and utilize this trust to implement secure anonymous payment.

9. Conclusion and Future Work

In this paper we presented the structure of a scalable and flexible service marketplace. The marketplace is realized by providing high level use cases represented by microflows, as well as low level microflows that describe the interaction between the high level use cases and actual services. Additionally the marketplace provides a set of properties for the selection / filtering of services.

As the presented marketplace contains no active functionality, but instead concentrates on the multi-level description of services it is a business enabler, but not a gateway or bottleneck. The feasibility of the marketplace (and its implementation) was shown by including real world services, in our case a diverse set of cinema services exposing searching and booking functionality.

Figure 6: Mobile Cinema Client

With the incarnation of the marketplace as a basis, we will concentrate on further elaborating more real world use cases and client applications. The real-world mobile cinema client (see Fig. 6), acts as a testbed to further the integration of secure payment [6], service specific SLA's and SLA-negotiation as outlined in Section 8.

References

[1] L. Schubert, K. Jeffery, B. Neidecker-Lutz, The future of cloud computing, Tech. rep. (2010).

[2] M. F. Greaver, Strategic outsourcing: a structured approach to outsourcing decisions and initiatives, AMACOM Div American Mgmt Assn, 1999.

[3] C. D. Huang, J. Goo, Rescuing IT outsourcing: Strategic use of Service-Level agreements, IT Professional 11 (1) (2009) 50–58.

[4] P. Lambros, M. T. Schmidt, C. Zentner, Combine business process management technology and business services to implement complex web services, IBM Corporation.

[5] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D. L. Mcguinness, E. Sirin, N. Srinivasan, Bringing semantics to web services with owl-s, World Wide Web 10 (3) (2007) 243–277.

[6] J. Mangler, C. Witzany, O. Jorns, E. Schikuta, H. Wanek, I. U. Haq, Mobile gSET - secure business workflows for Mobile-Grid clients, Concurrency and Computation: Practice and Experience 9999 (9999) (2009) n/a. doi:10.1002/cpe.1437.
URL http://dx.doi.org/10.1002/cpe.1437

[7] L. Clement, A. Hately, C. von Riegen, T. Rogers, et al., UDDI version 3.0. 2, http://www.uddi.org/pubs/uddi_v3.htm, [Last access: 02.08.2010] (2004).
URL http://www.uddi.org/pubs/uddi_v3.htm

[8] R. T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. thesis, University of California (2000).

[9] V. Cardellini, M. Colajanni, P. Yu, Dynamic load balancing on web-server systems, Internet Computing, IEEE 3 (3) (1999) 28–39. doi:10.1109/4236.769420.

[10] Relax NG home page, http://www.relaxng.org/, [Last access: 05.08.2010].
URL http://www.relaxng.org/

[11] J. Eder, J. Mangler, E. Mussi, B. Pernici, Using stateful activities to facilitate monitoring and repair in workflow choreographies, in: Proceedings of the 2009 Congress on Services - I, IEEE Computer Society, 2009, pp. 219–226.
URL http://portal.acm.org/citation.cfm?id=1591556

[12] O. Corcho, P. Alper, P. Missier, S. Bechhofer, C. Goble, Grid metadata management: requirements and architecture, in: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, 2007, p. 97–104.
[13] P. Missier, P. Alper, O. Corcho, I. Dunlop, C. Goble, Requirements and services for metadata management, IEEE Internet Computing 11 (5) (2007) 17–25.
[14] P. A. Bernstein, U. Dayal, An overview of repository technology, in: Proceedings of the International Conference on Very Large Data Bases, INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1994, p. 705–705.
[15] S. Jha, A. Merzky, G. Fox, Using clouds to provide grids with higher levels of abstraction and explicit support for usage modes, Concurrency and Computation: Practice and Experience.
[16] M. B. Blake, A. L. Sliva, M. zur Muehlen, J. V. Nickerson, Binding now or binding later: The performance of uddi registries, in: 40th Annual Hawaii International Conference on System Sciences, 2007. HICSS 2007, 2007, p. 171c–171c.
[17] R. Vigne, J. Mangler, E. Schikuta, C. Witzany, A RESTful repository to store services for simple workflows, in: Austrian Grid, National Symposium, OCG, Linz, Austria, 2009, p. 1.
[18] N. Srinivasan, M. Paolucci, K. Sycara, An efficient algorithm for OWL-S based semantic search in UDDI, Semantic Web Services and Web Process Composition (2005) 96–110.
[19] M. Risch, I. Brandic, J. Altmann, Using SLA mapping to increase market liquidity, in: A. Dan, F. Gittler, F. Toumani (Eds.), Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops, Vol. 6275, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 238–247.
URL http://www.springerlink.com/content/f6j07244581r7498/
[20] T. Weishaupl, C. Witzany, E. Schikuta, gSET: trust management and secure accounting for business in the grid, in: Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06, Vol. 1, 2006.