WEB SERVICE WRAPPING, DISCOVERY AND CONSUMPTION — MORE POWER TO THE END-USER

Keywords: web services, end-users, discovery, consumption, linked data

Abstract: B2B systems integration was revolutionised by the introduction of web services, transforming the way enterprise systems communicate and strengthening the relationships with providers and customers. The main advantage seen by companies was an increase of operational efficiencies, and a reduction of costs. In this scenario, highly qualified software developers are responsible for the integration of services with other systems, through the analysis of formal service descriptions or human-readable specifications. However, this model fails when targeting the long tail of enterprise software demand, the end-users. Discovery and consumption of web services are difficult tasks for end-users. This means that potential long tail of end-users creating task-specific applications from existing services is as of yet completely untapped. This paper presents an approach to facilitate the discovery and consumption of business web services by end-users, closing the gap between the two. The approach includes: (a) a method to generate ready-to-use web service wrappers, and (b) a catalogue which users can browse to search for web services fitting their needs.

1 Introduction

Business-to-business (B2B) integration is still a significant challenge, often requiring extensive efforts in terms of different aspects and technologies for protocols, architectures or security. Solutions based on open standards have reduced the complexity of integrating different business applications between different companies and partners. In this sense, Web services and XML standards such as RosettaNet, ebXML or OAG have been a boon to the world of B2B. By web services standards we are referring to the following open standards: Web Services Description Language (WSDL — to describe), Universal Description, Discovery and Integration (UDDI — to advertise and syndicate), Simple Object Access Protocol (SOAP — to communicate) and Web Services Flow Language (WSFL — to define work flows). A common scenario in their usage is a company using WSDL to describe its web services, publishing them through a repository using UDDI, while these web services use SOAP-based messages to achieve dynamic integration between different, disparate applications.

While the adoption of these standards has offered some advantages in business-to-consumer (B2C) integration as well, the interaction and consumption of web services still requires programming skills and a deep understanding of the technology, which poses an obstacle for an end-user with limited knowledge on the matter (in the sense as defined in (Fuchsloch et al., 2010)). Regarding the discovery of the right service for the right task — a crucial requirement for the dynamic use of WS — most publishing platforms are syntax-based, making it difficult to navigate through a large number of web services (Pilioura and Tsalgatidou, 2009), therefore preventing end-users from performing these tasks. Solutions such as Semantic Web Services (SWS) promised many advantages in discovery, composition and consumption of web services, independently of the provider's platform, location, service implementation or data format. However, as of yet they have not been widely adopted, possibly because the perceived potential benefits of SWS did not justify the additional investments required for businesses (Shi, 2007). In a similar vein as B2C, governments are currently opening their data to the public as linked data. So far, the efforts are restricted to

data only, so that users are left alone in finding applications of that data. A natural further development in the public sector would be that their services for administration tasks and single-window applications will be opened as well. Hence, governments will benefit from the research done around these concepts and technologies.

The motivation of our work has been strongly influenced by the end-users' needs. We are targeting users which are non-skilled in term of programming and software development, empowering them with a platform to discover and consume web services in a straight-forward manner. The web services and their specifications are not published right away in the platform, since this would simply mean to reimplement UDDI. However, their definition and behaviour are wrapped in what we call a "web service wrapper". The products resulting from the wrapping process are two artifacts: a specific definition of the web service to be used by this tool, and a ready-to-consume (inside a web browser) piece of code with the proper functions to invoke the web service.

The remainder of this paper is organised as follows. Section 2 covers the state of the art in web services publication, discovery and wrapping/consumption. In Section 3 we present our approach for web services wrapping, available possibilities and limitations. Next, Section 4 describes our platform for publishing web services and empowering end-users to discover services. Section 5 gives the reader a clearer use case and how this work is integrated into the FAST platform, and finally, we present our main conclusions and highlight future lines of research in Section 6.

2 Related work

Web services have been around for a long time since they first appeared. One of the most important claims about their benefits has been (syntactic) interoperability between third-party systems and applications based on different platforms / programming languages. System integration, within a company or between systems from different enterprises, became easier with the adoption of web service standard technologies, in particular the Web Service Description Language (WSDL), the most extensive standard used for the definition of web services. Many integrated development environments (IDEs) can deal with WSDL to facilitate the integration task, however it is still required that developers read and understand their specifications, and implement part of the logic programmatically. As a step forward, there are several tools which facilitate the interaction with data sources and services on the Web. Yahoo! Pipes, Apatar, JackBe Presto, Microsoft Popfly (now defunct) and NetVibes, among others provide a set of modules to access different kind of data sources, such as RSS feeds, a given web page (HTML code), Flickr images, Google base or the Yahoo! search engine, databases (MySQL, PostgreSql, Oracle), and powerful enterprise systems such as Salesforce CRM, SugarCRM and Goldmine CRM. However, none of the solutions in the market really facilitates end-users to build their own applications or widgets allowing the interaction with web services created by third-party providers. The solutions found are mainly data-oriented (RSS feeds, databases, raw text). Several tools permits some sort of (web) service integration, but are meant to be used within an enterprise level by savvy business users or developers. The solution presented in this paper leverage the possibility of integrating REST or SOAP-based web service inside visual applications running in a browser (e.g. widgets) by providing a platform to create, publish and discover web services wrappers.

In the context of publishing and discovering Web services, service providers have well-known and widely used technologies to accomplish the task of publication, such as the Universal Description, Discovery and Integration (UDDI) or WS-Discovery. The UDDI service registry specification (Clement et al., 2004) is currently one of the core standards in the Web service technology stack and an integral part of every major SOA vendor's technology strategy and offers to requesters the ability to discover services via the Internet. In short, UDDI serves as a centralised repository of WSDL documents. A similar concept is iServe (Pedrinaci et al., 2010). This platform aims to publish web services as what they called Linked Services — linked data describing services —, storing web service definitions as semantic annotations, so that other semantic web-aware applications may take advantage of it. However, the platform does not deal with the step from the definition to the consumption of the services.

A different approach for web service discovery is the Web Services Dynamic Discovery (WS-Discovery) specification (Beatty et al., 2005). The core of this approach is a multicast discovery protocol. Service providers and consumers listen to each other for new services specifications within a network, so there is no need of a centralised registry. As a drawback, WS-Discovery do not support Internet-scale discovery, making it useless for our purposes.

3 Wrapping web services

In our approach, wrapping web services is done in two steps: a first step is in charge of the construction of a service request and a second step analyses the response received from the execution of the service, allowing the analysis of response data and the mapping to domain-specific concepts. From this input, we generate JavaScript code to be embedded into web gadgets. This JavaScript code takes query parameters, constructs the service request, analyses the service response and returns result data in the desired format.

3.1 Constructing service requests

In this paper we illustrate the interaction with services on simple GET requests as supported e.g. by REST services. Support for other services like POST based services is under construction. In case of a simple GET request, a service request is assembled using a certain URL and a set of parameters. (A POST request would have a text block to be posted. In addition, other header data may be needed to transfer e.g. session ids.) As an example, the Ebay Shopping web service will be studied. As documented on the corresponding web site, the following URL is invoked to retrieve a list of items corresponding to the search keywords *USB*:

```
http://open.api.sandbox.ebay.com/shopping?
  appid=KasselUn-efea-4b93-9505-5dc2ef1ceecd&
  version=517&callname=FindItems&
  ItemSort=EndTime&QueryKeywords=USB&
  responseencoding=XML
```

As you may see, the URL invoked is `http://open.api.sandbox.ebay.com/shopping` and the parameters used in the example are:

appid this is the application ID obtained to use the API.

version the API version.

callname in this case FindItems to search through all items in Ebay.

itemsort sorting method for the list of items.

querykeywords list of keywords.

responseencoding format of the response message obtained by the invocation of the request.

The above URL for searching items in Ebay is followed by the query parameters, which take the form *argument=value*. In this example, the only relevant parameter the end user may want to provide is *querykeywords*. Thus the end user may use some input field to provide a query keyword and pass this value to our service wrapper operation. Other parameters can be set to a default value. However, in order to develop a more generic wrapper to a service, more or even all parameters might be passed by the user.

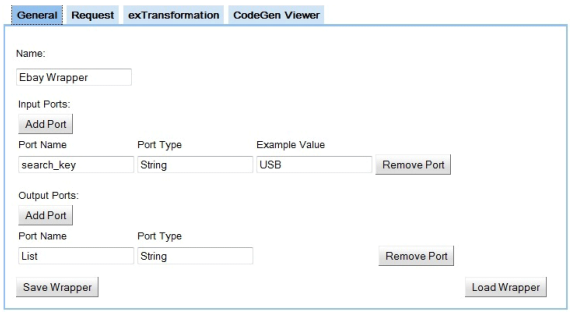

Figure 1: Configuring parameters of a service wrapper

To define the desired input parameters and the type of the expected result of a new service wrapper the service wrapper tool provides a web based form that allows the editing of these entries, see Fig. 1. Note, we allow to edit example values for the input parameters. These example values may be used to test the service wrapper.

The service wrapper tool composes the request using a template string which will contain placeholders for input values. Before sending the request to the service, the placeholders are replaced with their corresponding values and the resulting URL is then ready to be sent as the service request.

An example of service request construction can be found in the Figure 2. In the top input field the user may drop an example HTTP request taken e.g. form the service documentation e.g. from its website[1]. The tool analyses the example request and in the middle of the screen a form for editing the request parameters is provided. In our example, the user has connected the *QueryKeywords* parameter with the input parameter *search_key* by adding a corresponding reference to the value field of that parameter. In addition, we have retrieved an access key which has been entered as value for the *appid* parameter.

Below the request parameter editing form of Figure 2, a *Send Request* button allows to validate the constructed URL by sending it to the specified service address (via a server relay). Then the placeholders for input parameters are replaced by their example values and the resulting http request is shown below the parameter form. In addition, the request is sent and the response is shown on the bottom of that page. This gives the user a fast feedback whether the constructed request works as desired.

[1] http://developer.ebay.com/products/shopping/

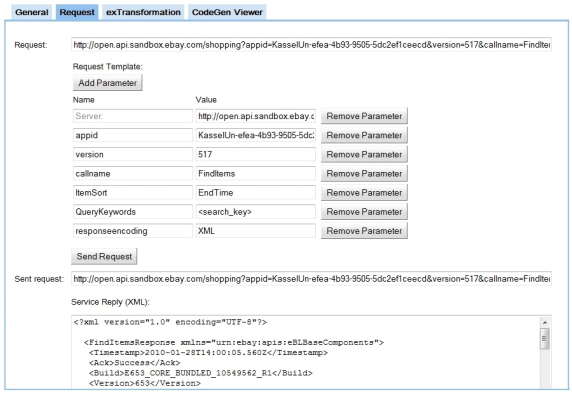

Figure 2: Constructing the service request URL and its parameters

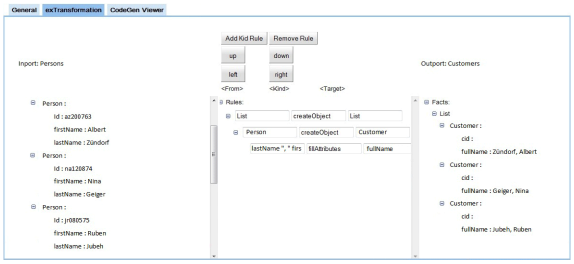

Figure 3: Interactive, rule based transforming of an XML response to domain objects

3.1.1 Limitations

It is worth pointing out that currently the wrapper tool is able to construct input ports for the wrappers using just basic types. To construct URL requests from more complex input parameters, as e.g. a person object, we are currently developing access operations that will allow to access the values of fields of complex objects. For example, the expression *customer.address* may refer to the *address* field of a *customer* parameter.

3.2 Interpreting service responses

The service request is constructed and sent to the service provider, who will then send back a response. This response message could be serialised in any format, though the most common formats used nowadays are XML or JSON. To continue the example started in the previous section, we assume the response of the Ebay Shopping service will be in XML as specified in the request.

3.2.1 Translation of XML into Facts

Once the service response, in XML format, has been retrieved, the transformation tab of the wrapper tool shows it as an interactive object tree, as seen on the left side of Fig. 3. To construct this interactive object tree, the XML document has been parsed into a DOM, and a simplified tree representation of that DOM is built up.

A transformation rule is used to analyse the XML data and to generate domain-specific objects from concepts from the ontologies used by the parameters of the different building blocks. A transformation rule is composed of three elements, as seen in the middle part of Fig. 3. First, the *from* field indicates the XML elements to be translated by the rule. These XML elements are identified by the tag name of a DOM element from the XML document. Second, the type of the rule will be set, taking one of the following values: *createObject*, *fillAttributes* or *dummy*. And third, the target of the rule specifies a certain concept or attribute, to be created or filled. A detailed explanation of the type of actions to be trigger from the transformation rules is as follows:

action *createObject* specifies the creation of a new domain object. The type of that new object is provided in the third compartment. In the example being explained, the root rule searches for XML elements with tag name *FindItemsResponse* and for each such element a *List* object is created. The resulting objects are shown in a facts tree in the right of Figure 3.

action *fillAttributes* does not create a new object but it fills the value of the attribute provided as third part of such rules. In our example, the third transformation rule searches for XML elements with tag name *Title*. Note, the rule is a sub-rule of the second rule, which generates *Product* objects. Thus, the sub-rule searches for *Title* tags only in the subtree of the XML data that has been identified by an application of the parent rule before. For example, the *Item* rule may just have been applied to the first *Item* element of the XML data. Then, the *Title* rule is applied only to the first *Item* sub-tree of the XML data and thus it will find only one *Title* element in that sub-tree (not visible in Figure 3). The value of that *Title* element is then transfered to the *productName* attribute of the corresponding *Product* object. Actually, our *from* fields allows also to refer to parts of an XML attribute e.g. to *words* 1 through 3. It is also possible to combine constant text and elements of multiple XML tree elements.

action *dummy* does not create or modify any objects

but such rules are just used to narrow the search space for their sub-rules. For example, in Amazon product data, the XML data for an item contains sections for *minimum price*, *maximum price*, and *average price*. Each such section contains the *plain price* and the *formatted price*. Thus, in the Amazon case, a rule that searches for *formatted price* elements within an *Item* element would retrieve three matches. Using a dummy rule, we may first search for *minimum price* elements and then search for *formatted price* elements within that sub-tree.

Our tool follows an interactive paradigm. Therefore, any time a change to a transformation rule is done, the transformation process is triggered and the resulting facts tree is directly shown. This process helps the user to deal with the slightly complex semantics of the transformation rules avoiding errors or mistakes. In addition, our tool is ontology-driven, therefore the service wrapper designer retrieves the domain-specific types from a backend server (such as the catalogue discussed in Section 4) together with the structure of each type, i.e. together with a description of the attributes of each object. Thus, the transformation rule editor is able to provide selection boxes for the target element of the rules. For a *createObject* rule, this selection box shows the object types available for that domain. For the *fillAttributes* rules, the selection box shows the attributes of the object type chosen in the parent rule. In addition, we may provide some analysis tool, which will help to guarantee that the objects generated by the transformation rules conform to the object types defined in the corresponding conceptual model (see Section 4.3). This helps to ensure that the objects generated by the designed service wrapper will be compatible for input parameters of subsequent filter steps and or gadgets.

3.3 Generating a Resource Adapter

Once the wrapping of a service has been defined and tested in the service wrapper tool, we generate an implementation of the desired Resource Adapter in XML, HTML, and JavaScript, ready to be deployed and executed inside a web gadget.

3.4 Limitations

The rule driven approach presented above is somewhat limited. It is deliberately restricted to such a simple rule mechanism in order to keep things simple enough for end-users. Still, the selected approach suffices for many practical and real world cases. As a more complex example, the XML data for a person may provide two different tags for the first and the last name of a person. Contrarily, a person fact which conforms to a certain ontology for that domain may provide only one *fullname* attribute that shall be filled by a concatenation of the first and the last name. To achieve this, the *from* field of that transformation rule might look like: `lastname"', "'firstname`. We are also able to do some navigation in the XML tree to follow XRef elements. For example the attribute `grandmother` could be filled using `mother.mother` in the *from* field.

However, there are some transformations that these rules cannot perform. For example, we do not support any mathematical operations. Thus, transforming e.g. Fahrenheit into Celsius temperatures is not supported. To cover such cases, intermediate object formats can be used which would allow generating objects to be further processed by additional filters. Such additional filters may be realised using (hand coded) operators, since some generic operators can act as filters for aggregation and conversions of objects from multiple sources. Then, service wrappers in combination with these filter operators will allow covering these complex cases.

4 Publishing and discovery platform

The area of web service publication and discovery has been subject for a lot of research since the very beginning the concept was coined. The current state of the art provides several solutions and strategies which providers and consumers may take advantage of. However, as explained in previous sections, a large number of them suffer from limited syntax-based descriptions and simple keyword-based search, while other more complex approach did not success to be widely spread regarding the few benefits offered compared to their drawbacks. Moreover, the gap between discovery and consumption is an obstacle for their usage by non-technical end-users. In order to improve this issue, this paper presents a novel publishing platform, permitting any enterprise or individual to publish their public domain web services, with an enhanced semantic search based on the definition of the services, and serving web service wrappers for easy consumption in web applications.

4.1 Overview

The main difference of this platform with regards to the state of the art is that it is targeting a different kind of user, and covers many deficiencies others tools do

not tackle. As a brief overview, the platform being presented:

- allows functional discovery through web service pre- and post-conditions;
- serves web services wrappers ready to consume in web applications;
- provides advanced search capabilities, based on the formal service definition and inferences extracted from it;
- supports managing its resources via its RESTful API;
- offers a SPARQL endpoint, giving direct access to the data through complex queries;
- offers web service descriptions as linked data;
- follows well-known best practices for publishing data on the web;
- supports content negotiation so that different clients may retrieve information in their preferred format, choosing from JSON, RDF/XML, RDF/N3 or a human-readable HTML version.

This platform was intended and it is being used as part of the storyboard tool of the FAST platform (Hoyer et al., 2009), and it communicates with other components through its RESTful API, using JSON for the input of the requests. This paper will focus on the functionality regarding web service publication and discovery, and the process to publish any third-party web service into the platform. This process requires two steps: (i) generation of the wrapper (see Section 3), and (ii) creation of the resource in the publishing platform. The following is an example of part of a request sent to create a new wrapper into the platform (for the sake of clarity, some parts of the request have been omitted, but the main idea is shown):

```
{
  "code": "http://fast.morfeo-project.org/amazon-search.js",
  "creationDate": "2010-01-26T17:01:13+0000",
  "description": {"en-gb": "Amazon web service - Search"},
  "label": {"en-gb": "Amazon search"},
  "actions": [{
    "name": "filter",
    "preconditions": [[{
      "id": "item",
      "label": {"en-gb": "Ebay List"},
      "pattern": "?item
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type
        http://aws.amazon.com/AWSECommerceService#Item"
  }]],
  "postconditions": [
    ...
  ],
  ...
```

The enriched search capabilities are supported by the definition of pre- and post-conditions. They allow to define the inputs and outputs of the services and other building blocks using concepts from any ontology, and in this way to find web services or other building blocks which match each other and can therefore be integrated. The concepts of pre-/post-conditions were strongly influenced by WSMO (Roman et al., 2005), simplified and implemented in RDFS for better live performance. The model used to define the web service wrappers and the ontology created are described in more detail in Section 4.3.

4.2 Platform architecture

The architecture comprises an RDF store used for persistence, a business layer dealing with the model and reasoning, and a public facade providing the core functionality as a RESTful API, as well as a SPARQL endpoint accessing the RDF store directly. This presentation layer is aimed to interact with the FAST Gadget Visual Storyboard (see Section 5), or any other third-party application.

The RESTful API and the SPARQL endpoint are part of the presentation layer. The SPARQL endpoint is offered using the SPARQL protocol service as defined in the SPROT specification (Grant et al., 2008) and is aimed to enable third-party developers to directly query the knowledge base using SPARQL queries. This feature is supported by the Sesame RESTful HTTP interface for SPARQL Protocol for RDF.

The business logic layer contains all the domain-specific processing and reasoning. It provides functions to interact with all the elements of the domain model specified in 4.3, acting as a mediator between the public RESTful API and the persistence layer.

The persistence layer provides an API allowing to work with a standard set of objects representing the model. The interaction with the underlying RDF store is made via RDF2Go, an abstraction over triple (and quad) stores, which allows to program against RDF2Go interfaces and choose any RDF store implementation. This allows having a completely extensible and configurable framework for storage mechanisms, inferencers, RDF file formats, query result formats and query languages.

4.3 Conceptual Model

The conceptual model used to define the web service wrappers within this application has been influenced by WSDL, the main web service definition language, and semantic approaches such as WSMO and OWL-

S. It belongs to a more complex conceptual model for FAST, which can be grouped into three levels: the gadget and screenflow level at the top, the level of individual screens in the middle, and the level of web services at the bottom. In short, a screenflow is an executable (in a Web browser) set of screens, while a screen is a set of building blocks with a specific purpose. These building blocks — *forms* for the UI, *operators* for data manipulation and *back-end services* — all share the same structure. Every building block has a set of actions (operations in WSDL), each of which needs a set of pre-conditions (inputs) to be fulfilled in order to be executed, and provides a set of post-conditions (outputs) to other building blocks. These pre- and post-conditions are defined as individual RDF triples. This means that conditions can be modelled as graph patterns using SPARQL notation (Prud'hommeaux and Seaborne, 2008), where e.g. the post-condition of a login service such as "there exists a user" will be transformed into a simple pattern like "a variable `?user` is of type `sioc:User`", or "`?user a sioc:User`". Using this mechanism, extended by RDFS entailment rules, services and other building blocks can be matched automatically. The publishing and discovery platform (catalogue) can employ this functionality to support the discovery of web services based on the current user needs (expressed in the same way as pre-conditions). For further details on the conceptual model, see its definition in (Möller et al., 2010).

4.4 Discovery mechanisms

The platform presented in this paper is intended to support web service discovery and recommendation capabilities. As explained in detail in Section 4.3, there are different types of building blocks which can be modelled using the platform, leading to a necessity for different kinds of discovery or recommendation mechanisms, depending on the level of composition to deal with: screen-flow or screen. In the screen-flow composition, the smallest functional unit is the screen, meaning that a screen-flow is composed by interconnecting a number of screens between each other. This composition can be applied to any sort of building block since they all share the same structure: a set of pre- and post-conditions, which make possible the communication with other building blocks. That said, technically the algorithms presented in this section could be used for the discovery and composition of any type of building block stored in the platform.

At the screen-flow design level, the composition is made using a set of screens, and the pre- and post-conditions of the screen-flow itself used as entry and exit points of the screen-flow. A screen is only reachable when its pre-conditions are satisfied as well, meaning that the post-conditions of another screen, or the pre-conditions of the screen-flow are compatible with them.

The main goal of the discovery process is to help the user to make the screen-flow executable, in other words, to make all the screens within a screen-flow reachable. The platform offers two mechanisms to find and recommend screens, previously stored in the building block base (the catalogue), which will make the screen-flow executable: a simple discovery based on pre- and postconditions, and a multi-step discovery or planning algorithm. Like any other search engine or recommender system, the results are ranked before being presented to the user. The ranking algorithm is covered in Section 4.4.3.

4.4.1 Simple discovery based on pre- and post-conditions

At a particular state of a composition, ie. at screen-flow design, several of the building blocks used to compose it, screens in the case of a screen-flow, might be unreachable, therefore there must be pre-conditions not satisfied. The platform will assist the process by recommending building blocks which will satisfy these pre-conditions. In a nutshell, it collects the pre-conditions of all the unreachable building blocks, used as a graph pattern (see Section 4.3) in order to be matched against the post-conditions of any building block. In the following scenario, there are two screens: *s1* and *s2*. *s1* has as a pre-condition: "there exists a search criteria", and as a post-condition: "there exists a item". *s2* has just a pre-condition stating: "there exists a search criteria". Here you can see the same example but formally defined:

```
:G1 { :s1 a fgo:Screen .
      :s1 fgo:hasPrecondition c1 .
      :s1 fgo:hasPostcondition c2 .
      :c1 fgo:hasPattern GC1 .
      :c2 fgo:hasPattern GC2 .
      :s2 a fgo:Screen .
      :s2 fgo:hasPrecondition c3 .
      :c3 fgo:hasPattern GC3 }
:GC1 { _:x a amazon:SearchCriteria }
:GC2 { _:x a amazon:Item }
:GC3 { _:x a amazon:SearchCriteria }
```

The algorithm will construct a SPARQL query to retrieve building blocks satisfying the pre-conditions *c1* and *c3*. The query, although simplified for the sake of clarity, would look something like:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fgo: <http://purl.oclc.org/fast/ontology/gadget#>
```

```
PREFIX amazon:<http://aws.amazon.com/AWSECommerceService#>

SELECT DISTINCT ?bb
WHERE {
  ?bb rdf:type fgo:Screen .
  {
    {
      ?bb fgo:hasPostCondition ?c .
      ?c fgo:hasPattern ?p .
      GRAPH ?p { ?x rdf:type amazon:SearchCriteria }
    }
    UNION
    {
      ?bb fgo:hasPostCondition ?c .
      ?c fgo:hasPattern ?p .
      GRAPH ?p { ?x rdf:type amazon:Item }
    }
  }
  FILTER (?bb != <http://fast.org/screens/S1>)
  FILTER (?bb != <http://fast.org/screens/S2>)
}
```

However, in the example presented, it is straightforward to spot that the pre-condition *c3* would be satisfied whether the screen *s1* was reachable, being unnecessary to query the recommender algorithm to retrieve screens satisfying this pre-condition, hence these pre-conditions are removed for the construction of the query reducing the complexity of the problem, hence improving the overall performance of the algorithm. Moreover, it leads to find better results for the user, since the focus is given to make satisfy the pre-conditions which could not be satisfied by any of the elements of the current composition.

4.4.2 Enhanced discovery: search tree planning

In articial intelligence, the term *planning* originally meant a search for a sequence of logical operators or actions that transform an initial world state into a desired goal state. Presently, it also includes many decision-theoretic ideas, imperfect state information, and game-theoretic equilibria.

This paper applies the concept of planning to the design of building blocks. Back to the screen-flow design, the goal would be to make the screen-flow executable; the initial state would be a certain screen which is not yet reachable, and the plans would be sets of screens which are reachable by themselves, and accomplish the goal. The algorithm has been influenced by the ideas behing the heuristic search. For a certain state, i.e. the initial state which contains the screen to make reachable, a large tree of possible continuations is considered, in fact any screen would fit. Those screens which post-conditions do not satisfy in any form the unsatisfied pre-conditions are discarded, reducing the branches of the tree. A branch stops growing when a screens is already reachable (i.e. it has no pre-conditions) becoming a leaf of the tree. Once there are not screens added in a certain step, the algorithm stops and discards the incomplete branches, in other words, those branches where the leaf is not reachable.

It is worth pointing out some of the tree structure is pre-computed beforehand to speed up the querying process at runtime. Any time a building block is inserted into the catalogue, the algorithm is executed following two approaches: *forward search* and *backward search*. The forward search approach finds the building blocks whose pre-conditions will be satisfied by the post-conditions of the new building block while the backward search finds the building blocks whose post-conditions will satisfy the pre-conditions of the recently created building block. This data is stored and used for the tree or plan builder algorithm, without needing to check the compatibility of pre- and post-conditions for every single building block at every request.

4.4.3 Results ranking

Previous sections presented different mechanisms to search or discover building blocks which may satisfy the user needs. The results ensure compatibility based on the functional specification of the building blocks, however they are not ranked, and their quality is not measured. This section explains the ranking techniques applied for the different discovery mechanisms.

The ranking algorithm for 4.4.1 applies the following rules:

1. it gives a higher position to those building blocks which satisfy the highest number of pre-conditions;
2. it prioritise building blocks created by the same user who is querying;
3. it adjust the rank by using the ratings given to the building blocks, and their popularity in terms of usage statistics;
4. it weights the results according to non-functional features such as availability. This is only applied for what we call “web service wrapper”, and it is calculated periodically by invoking the wrapped web services generating an up-time rate.

For the planning case, the objective is not only to produce a plan but also to satisfy user-specified preferences, or what is known as *preference-based planning*. The ranking algorithm:

1. minimizes the size of the plans, after removing the elements of the plan which are already in the

canvas, so it gives priority to the elements the user has already inserted,

2. adjust the rank by using the rules 2, 3 and 4 used from the ranking algorithm of simple discovery based on pre- and post-conditions.

4.5 Serving Linked Data

The recently success of the Web of Data has influenced the way information is now published on the web, and has been widely adopted by the academic community. Also large companies such as The New York Times and BBC, and national governments such as US and UK made public commitments toward open data. The main aim of the linking data is to find other related data based upon previously known data, in terms of the connections or links between them. We apply this concept in order to provide metadata about the web services as linked data, following the principles as defined in (Bizer et al., 2009), in order to make them available to arbitrary third-party applications. Each web service is identified by an HTTP URI and hosted in the publishing platform so that it can be dereferenced through the same URI. For each building block, data is available in representations in different standard formats such as JSON (for communication with the applications such as the widget designer shown in Fig. 4), RDF/XML, Turtle, or even HTML+RDFa as a human-readable version. The principles and best practices proposed in (Berrueta et al., 2008) and (Sauermann et al., 2008) are also taken into account, so that these representations are served based on the request issued by the requesting agent, using a technique called *content negotiation*. As suggested by linked data principles, individual building blocks link to other data on the web, thereby preventing so-called isolated "data islands".

To allow a third-party application to retrieve the content in the format required, we support content negotiation as stated in the (Fielding et al., 1999), serving the best variant for a resource, taking into account what variants are available, what variants the server may prefer to serve, what the client can accept, and with which preferences. In HTTP, this is done by the client which may send, in its request, accept headers (`Accept`, `Accept-Language` and `Accept-Encoding`), to communicate its capabilities and preferences in format, language and encoding, respectively. Concretely, the approach followed is agent-driven negotiation, where the user agent selects the specific representation for a resource.

5 Use case

The intention of this paper was to show the advances made regarding web service discovery and consumption through the two artifacts developed as part of our research. While these artifacts may be deployed and used separately by third-party applications, they were originally developed to form the backbone of the FAST platform. FAST constitutes a novel approach to application composition and business process definition from a top-down user-centric perspective. Therefore, it is aimed at allowing users (mainly business users at the management level, without programming experience) to create their own situational applications by visually combining different components, or building blocks, such as graphical forms and back-end services in terms of their inputs and outputs. The work covered in this paper are the components highlighted and surronded by a dashed frame in the Fig. 4. These components have public interfaces to communicate with any application developed by a third-party developer or company, and as described in Sects. 3 and 4, are fully integrated and present a high cohesion with the rest of the components of the FAST architecture.

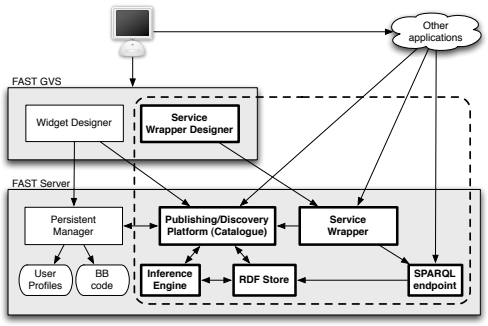

Figure 4: Overview of the FAST architecture

In order to facilitate a better understanding of the use case, we will describe a number of concepts related to FAST in this section. A *gadget* is the end product of the platform, aimed to be used in a specific mashup platform. Within FAST, the platform-independent gadget is called *screenflow*, which comprises a set of domain-specific *screens* connected through their *pre-* and *post-conditions*. A screen is the most complex building block fully functional by itself. It is composed by a *form* conveying the graphical interface, and a set of *operators* and *back-end services*, wrapped into so-called *service resources*.

The web service wrapper tool is then used to create the formal definitions of back-end services, trans-

forming them into building blocks to be stored and reused within the FAST platform. The tool is integrated directly into the graphical IDE, and for the storage interacts with the catalogue, where the formal service descriptions and the executable code encapsulating the service behaviour is stored.

The publishing platform presented, also called "Catalogue" within the FAST platform, covers several important purposes:

Publication and search of gadgets The output of the FAST IDE are gadgets, which are self-contained and ready to use. They need to be made available for download, as well as being discovered according to their formal description.

Publication and search of building blocks Every single component or building block definition is stored in the catalogue, which facilitates searching, recommendation and planning (the suggestion of combinations of building blocks to fulfil certain requirements).

Storage and indexing of user profiles and histories Each user of the FAST platform has a representation in the catalogue, thus making it possible for the IDE to make informed guesses about what they want in a particular situation, based on their previous behaviour.

Facilitate ontology mediation Because FAST allows the integration of third-party external services, it is not always guaranteed that all the data coming from a service is directly compatible with the FAST platform. In such cases, mediation between the different ontologies is necessary. The catalogues therefore provide additional mediation functionality (first steps in this direction are outlined in (Ambrus et al., 2009)).

6 Conclusions and future work

It has been demonstrated that adopting web services standards lead enterprises to increase operational efficiency, reduce costs and strengthen the relations with partners. Many of these standards have been widely adopted, such as WSDL, XML and UDDI, and many companies offer access to their information and operational systems through web services. However, when dealing with end-users, the process for publication and consumption is not well defined. The common way of publishing services is by providing some sort of definition on their websites, such a WSDL document and further details in a human-readable HTML page. Discovery is performed through a search engine or aggregator, mainly in a syntax-based way. For consumption, in most cases, consultants with good programming skills are required. In a move towards improving this situation, the work presented in this paper empowers the end-user with a platform (the *catalogue*) to easily discover services based on functional behaviour (pre-/post-conditions) and other metadata, being able to download a so-called resource adapter allowing the consumption of web services using standard languages to execute within a web browser, and a tool to transform, in an interactive manner, formal definitions of web services into these resource adapters, ready for being publisher into the catalogue.

The current version of the wrapping tool permits to create resource adapters for RESTful web services. As a next step, we will tackle SOAP-based web services described in WSDL documents. Once this is done, we will cover the two major paradigms for defining web service interfaces. However, we also consider to include semantically enriched WSDL documents using SAWSDL, and to support other SWS approaches such as WSMO-lite services.

Another limitation of the platform is inherent in web client technologies and the cross-domain policy for security. This is solved within FAST using platform-dependent API calls. Hence, the code generated makes use of the FAST API, which will be transformed depending on the mashup platform the user intends to deploy the gadget. To avoid these issues, and to be able to provide platform-independent code right away, we are planning to deploy the JSONRequest approach as proposed by (Crockford, 2009) and other solutions.

REFERENCES

Ambrus, O., Möller, K., and Handschuh, S. (2009). Towards ontology matching for intelligent gadgets. In *Workshop on User-generated Services (UGS2009) at ICSOC2009, Stockholm, Sweden*.

Beatty, J., Kakivaya, G., Kemp, D., Kuehnel, T., Lovering, B., Roe, B., John, C. S., (Editor), J. S., Simonnet, G., Walter, D., Weast, J., Yarmosh, Y., and Yendluri, P. (2005). Web services dynamic discovery (WS-Discovery). Specification, Microsoft.

Berrueta, D., Phipps, J., and Swick, R. (2008). Best practice recipes for publishing rdf vocabularies. Technical report, W3C.

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data - the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*.

Clement, L., Hately, A., von Riegen, C., and Rogers, T. (2004). Uddi version 3.0.2 specification. `http://uddi.org/pubs/uddi_v3.htm`.

Crockford, D. (2009). JSONRequest. Proposal.

Fielding, R. T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol – HTTP/1.1.

Fuchsloch, A., Janner, T., Hoyer, V., Urmetzer, F., Jubeh, R., and Fernández, R. (2010). Deliverable d6.1.2: Scenario definition. Deliverable, FAST Consortium.

Grant, K., Feigenbaum, L., and Torres, E. (2008). SPARQL protocol for RDF. Recommendation, World Wide Web Consortium (W3C).

Hoyer, V., Janner, T., Delchev, I., López, J., Ortega, S., Fernández, R., Möller, K. H., Rivera, I., Reyes, M., and Fradinho, M. (2009). The FAST platform: An open and semantically-enriched platform for designing multi-channel and enterprise-class gadgets. In *The 7th International Joint Conference on Service Oriented Computing (ICSOC2009), Stockholm, Sweden.*

Möller, K., Rivera, I., Ureña, M. R., and Palaghita, C. A. (2010). Ontology and conceptual model for the semantic characterisation of complex gadgets. Deliverable 2.2.2, FAST Project (FP7-ICT-2007-1-216048).

Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecký, J., and Domingue, J. (2010). iServe: a linked services publishing platform. In *Workshop: Ontology Repositories and Editors for the Semantic Web at 7th Extended Semantic Web Conference*.

Pilioura, T. and Tsalgatidou, A. (2009). Unified publication and discovery of semantic web services. *ACM Trans. Web*, 3(3):1–44.

Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL query language for RDF. Recommendation, W3C. `http://www.w3.org/TR/rdf-sparql-query/`.

Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., and Fensel, D. (2005). Web service modeling ontology. *Applied Ontology*, 1(1):77–106.

Sauermann, L., Cyganiak, R., Ayers, D., and Völkel, M. (2008). Cool URIs for the Semantic Web. Interest group note, W3C. `http://www.w3.org/TR/cooluris/` 05/05/2009.

Shi, X. (2007). Semantic web services: An unfulfilled promise. *IT Professional*, 9:42–45.