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

Ismael Rivera, Knud Hinnerk Möller
*DERI, National University of Ireland, Galway*
*ismael.rivera@deri.org, knud.moeller@deri.org*

Albert Zündorf
*University of Kassel, Germany*
*zuendorf@cs.uni-kassel.de*

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 such as RosettaNet, ebXML, the Web Service Description Language (WSDL), Universal Description, Discovery and Integration (UDDI) or the Simple Object Access Protocol (SOAP) have reduced the complexity of integrating different business applications between different companies and partners.

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.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 re-implement 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.

## 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.

## 3 Context: The FAST Platform

The intention of this paper is to demonstrate the advances made regarding web service discovery and consumption through two artifacts developed as part of our research: the *publishing and discovery platform* and the *service wrapper tool*. 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 (Hoyer et al., 2009). FAST constitutes a novel approach to application composition and business process definition from a user-centric perspective. It is aimed at allowing users without previous programming experience to create their own situational applications by visually combining different *building blocks*, such as graphical forms and back-end services, based on their inputs and outputs (or *pre-* and *post-conditions*). The work covered in this paper are the components highlighted and surrounded by a dashed frame in the Fig. 1. Communication within the platform is mostly done via a RESTful API, using JSON as an exchange syntax.

In order to facilitate a better understanding of these components, we describe a number of concepts related to FAST in this section. A *gadget* is the end product of the platform, ready to be run in any ordinary web browser, usually through a 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. Visually, it is

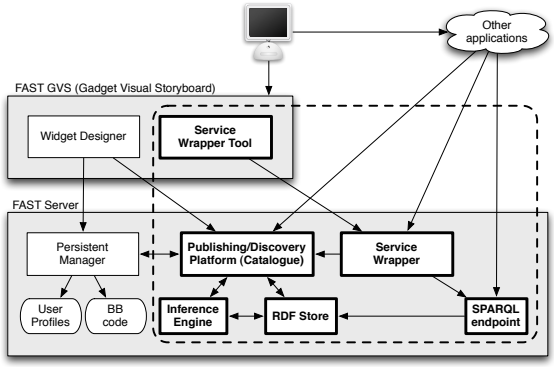


Figure 1: Overview of the FAST architecture

similar to a tab in a tabbed application. 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 service wrapper tool is used to create wrappers for third-party web services, transforming them into building blocks to be stored and reused within the FAST platform. The publishing platform presented, also called *catalogue* within the FAST platform, covers several important purposes, such as storage, indexing, publication and search of gadgets, gadget building blocks and user profiles, as well as some level of ontology mediation to facilitate the integration of services from different parties (first steps in this direction are outlined in (Ambrus et al., 2009)).

# 4 Publishing and Discovery Platform

As explained in previous sections, current solutions and strategies for web service publication and discovery suffer from limited syntax-based descriptions and simple keyword-based search, while other more complex approach failed because the added complexity did not offer sufficient benefits. This paper therefore presents a novel publishing platform (the *catalogue*), permitting any enterprise or individual to publish their public web services, providing enhanced semantic search service wrappers for easy consumption in web applications.

## 4.1 Overview

One of the main difference of this platform with regards to the state of the art is that it is targeting a different kind of user. 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.

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 can 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 catalogue 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 Fig. 1), or any other third-party application.

## 4.2 Conceptual Model

The conceptual model used to define the web service wrappers within this application has been influenced by both WSDL and semantic approaches such as WSMO and OWL-S. It is part of a more complex conceptual model for FAST (Möller et al., 2010), 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. All these building blocks and their subparts share the same structure: a set of actions (like 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 RDF graph patterns. E.g., the post-condition of a login service such as "there exists a user" will be expressed as a simple pattern such as "`?user a sioc:User`"[1]. Using this mechanism, extended by RDFS entailment rules, services and other building blocks can be matched automatically. The publishing and discovery platform can employ this functionality to support the internal discovery of web services based on the current user needs (expressed in the same way as pre-conditions).

## 4.3 Discovery Mechanisms

The main goal of the discovery process is to aid the user in finding suitable building blocks to complement the ones they are already using. E.g., on the screen-flow level, this would mean to suggest additional screens to make existing screens within a screen-flow reachable, and the screen-flow therefore executable. The platform offers two mechanisms to find and recommend screens (or other building blocks) stored in the catalogue: a simple discovery based on pre- and postconditions, and a multi-step discovery or planning algorithm. Before being presented to the user, the results are being ranked, as discussed in Sect. 4.3.3.

### 4.3.1 Simple Discovery Based on Pre- and Post-Conditions

In this simple approach, the platform will assist the process by recommending all building blocks which will satisfy currently unfulfilled pre-conditions. The pre-conditions of all the unreachable building blocks are collected as a graph pattern, which is then matched against the post-conditions of all available building blocks. 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".

```
: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 }
```

[1]We use Trig (`http://www4.wiwiss.fu-berlin.de/bizer/TriG/`) and SPARQL (`http://www.w3.org/TR/rdf-sparql-query/`) notation for RDF graphs throughout this paper.

```
: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.

### 4.3.2 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.

#### 4.3.3 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 4.3.1 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.

### 4.4 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. 1), 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 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.

## 5.1 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.

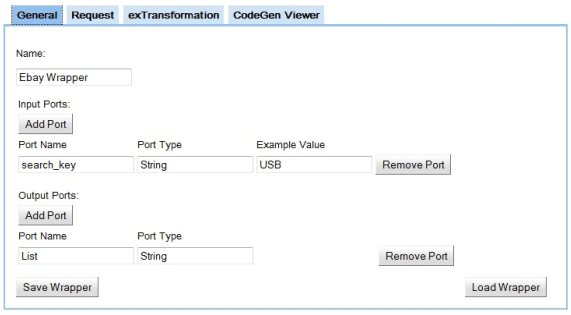


Figure 2: Configuring parameters of a service wrapper

**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.

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. 2. Note, we allow to edit example values for the input parameters. These example values may be used to test the service wrapper.

An example of service request construction can be found in the Figure 3. In the top input field the user may drop an example HTTP request, taken e.g. from the service's documentation[2]. 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 3, a *Send Request* button allows to validate the constructed URL by sending it to the specified service address (via a server relay). The service's response is shown on the bottom of the page. This gives the user

[2] `http://developer.ebay.com/products/shopping/`

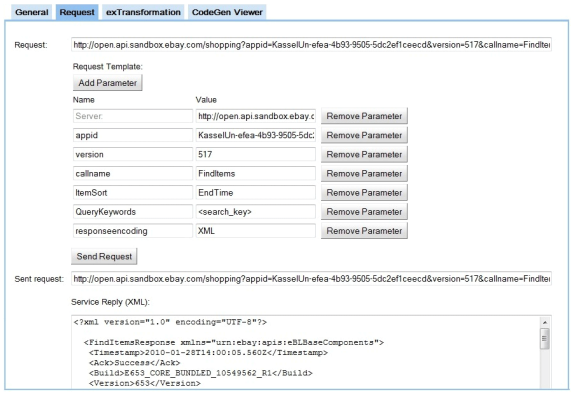


Figure 3: Constructing the service request URL and its parameters

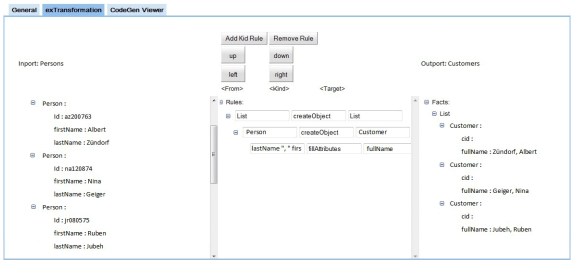


Figure 4: Interactive, rule based transforming of an XML response to domain objects

a fast feedback whether the constructed request works as desired.

### 5.1.1 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.

## 5.2 Interpreting Service Responses

A web service can choose to send a response message in any desired structured format. In our example, we assume an XML tree, but a JSON structure is equally possible.

### 5.2.1 Translation of the Response Tree into Facts

Once the service response 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. 4.

A transformation rule is used to analyse the response 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. 4. First, the *from* field indicates the source elements to be translated by the rule. 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:

**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 4.

**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 4 ). The value of that *Title* element is then transfered to the *productName* attribute of the corresponding *Product* object.

**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.

### 5.3 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.

### 5.4 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.

## 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.

## ACKNOWLEDGEMENTS

This work is supported in part by the European Commission under the first call of its Seventh Framework Program (FAST STREP Project, grant INFSO-ICT-216048) and in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Líon-2).

## 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*.

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.

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.

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.