FAST AND ADVANCED STORYBOARD TOOLS

FP7-ICT-2007-1-216048

http://fast.morfeo-project.eu

Deliverable D4.3.1

Mechanisms for Gadget-Service Connections and Gadget Functionality

Ismael Rivera, NUIG

Albert Zündorf, UniKassel

Date: 26/02/2010

FAST is partially funded by the E.C. (grant code: FP7-ICT-2007-1-216048).

Version History

Rev. No.	Date	Author (Partner)	Change description
1.0	26.02.2010	Ismael Rivera (NUIG)	Final version ready for external review

Executive Summary

This deliverable exposes several mechanisms which will allow the connection and interaction between end-user's interfaces to third-party back-end services.

These back-end services cannot be directly used; hence they need to be encapsulated or wrapped into the so-called Resource Adapters within the FAST platform. The application of semantics to the back-end, through the corresponding resource adapters, and front-end building blocks ensures a powerful instrument in the task of building new gadgets, improving the search, and enhancing the connection among the different building blocks which compose a gadget.

Therefore, the focus of this deliverable is to define how these wrappers will be constructed to allow the FAST platform to exploit web services [Alonso et al., 2003] such as RESTful web services, SOAP-based web services and semantic web services through WSMO, and define mechanisms to connect them within the gadgets.

Document Summary

Code	FP7-ICT-2007-1-216048	**Acronym**	FAST
Full title	Fast and Advanced Storyboard Tools		
URL	`http://fast.morfeo-project.eu`		
Project officer	Annalisa Bogliolo		

Deliverable	**Number**	D4.3.1	**Name**	Mechanisms for Gadget-Service Connections and Gadget Functionality
Work package	**Number**	4	**Name**	Visual composition of screen-flow resources and interoperability with back-end Web Services

Delivery data	**Due date**	28/02/2009	**Submitted**	27/02/2009
Status			final	
Dissemination Level	Public ☒ / Consortium ☐			
Short description of contents	This deliverable presents techniques to exploit different kind of web services such as RESTful, SOAP-based and WSMO services. It exposes how these services are commonly defined, and what FAST need to allow these web services to be use within a gadget. In this deliverable is also presented a tool which apply some of these techniques allowing users to create the necessary wrappers for these services.			
Authors	Ismael Rivera, NUIG Albert Zündorf, UniKassel			
Deliverable Owner (Partner)	Ismael Rivera, NUIG		**email**	ismael.rivera@deri.org
			phone	+353 91 495338
Keywords	FAST, web service, WSDL, REST, SOAP, WSMO, Resource Adapter			

Table of contents

List of Figures

1 Introduction

1.1 Goal and Scope

The objective of this deliverable is the analysis and development of mechanisms to facilitate the connection between gadgets and underlying Web services, based on front-end user requirements and on the semantic descriptions of the service wrappers, and the development of these service wrappers from the Web services' APIs and formal descriptions.

1.2 Structure of the Document

This deliverable is structured as follows: Section 1 states the goal, scope and structure of the document. Before going to the core of the document, a brief introduction to some related work is done in Section 2. Then, Section 3 describes what a resource adapter is, how the platform is able to interact with web services, how these wrappers or adapters are build and then discovered to be reused in any gadget. Finishing the deliverable, Section 4 defines the service wrapper tool built to facilitate FAST users to create resource adapters of RESTful web services.

2 Related Work

This section focuses on different approaches dealing with web services integration and reutilisation. That is not a novel idea, since one of the basic pillars of web services is to allow any application to access and execute third-party web services over a network, and there are already loads of approaches about this issue. However, most of these approaches require a high-level technical profile to accomplish this integration in a programmatically manner. The ones studied in this deliverable are those which presents and allow reusing data sources and web services using a graphical interface or wizard.

[Urmetzer et al., 2010] presented several tools to create widgets, which allow the interaction with some kind of data sources and services. Yahoo! Pipes provides a set *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, among others. These modules are developed by the widget platform itself, they do not allow users to build their own modules, and it lacks of a way to interact with REST or SOAP web service. Another tool to reference is Apatar. Apatar is more enterprise-oriented, allowing the interaction with databases (MySQL, PostgreSql, Oracle), and powerful enterprise systems such as Salesforces.com or SugarCRM. It also provides access to data sources such as XML or text files. But again, there is no way to use web services. Finally, Presto is another solution enterprise-oriented. It allows the common data sources such as RSS and ATOM feeds or XML-based sources. However, Presto brings connectors to use services from HP SOA Systinet and any Oracle information technologies including Oracle 9i/10g/11g, Oracle Fusion SOAs, and Oracle Applications.

Summarising up, none of the solutions in the market really facilitate the users to build up their own widgets using any kind of web service in a graphical manner. Their approaches are either data-oriented (RSS feeds, databases, raw text), or including a small set of web services in the case of Presto. FAST will bring the possibility of integrate REST or SOAP-based web service inside a gadget (by creating service wrappers), and graphical tool will be provided to facilitate the process of building these service wrappers. In a similar way, and in the context of bioinformatics web services, [Gordon and Sensen, 2008] explains how to publish web services to be reused in their tool as Moby (semantic) services. They have developed the Daggoo[1] prototype which will create these service wrappers to be used in Moby.

[1] http://www.daggoo.net/

3 Resource Adapters

In the context of FAST philosophy, a service is a software element or system, often deployed within enterprise boundaries, designed to support interoperable Machine-to-Machine interaction over a network (e.g. Web Services, Databases, CORBA or RPC interfaces...). These services have to be used by the gadgets, but there are several complications to be resolved in order for the front-end (i.e. the gadget user interface and logic) to communicate with those services, such as the disparity of interfaces and invocation mechanisms. Another issue is the complexity for an end-user to interact with those services. Hence, this interaction will be shifted to a user-interaction paradigm through a user-friendly graphical interface (service front-end), allowing both humans and machines to interact with the services through a uniform fashion.

The following sections explain what a resource adapter is, how a user, in this case a resource developer, can build these resource adapters, and how another user, a screen developer for instance, is able to discover and connect them while creating a screen. Hence, these three phases are called: building, discovery and connection.

3.1 Definition

A complex gadget in terms of FAST is aimed to provide a functional access from a graphical user interface to a set of services (SOAP or REST-based Web services) and data sources (Atom/RSS feeds). These services and data sources need to be modelled and encapsulated within the platform in order to allow being discovered and used by other components (i.e. forms and operators). These service wrappers are called Resource Adapters in the FAST platform. Figure 1 illustrates the role they play and which other building blocks they interact with within a gadget.

Basically, a resource adapter, from a design point of view, is composed of:

1. **metadata** such as creator, label, description, version, and so on,
2. a set of **actions** or operations, with their corresponding preconditions,
3. and a set of **postconditions** or outputs of the resource.

Figure 2 shows the information above in a graphical manner. An action (represented by a white square) is the minimal unit of functionality a resource may have. Imagine a user clicking on a

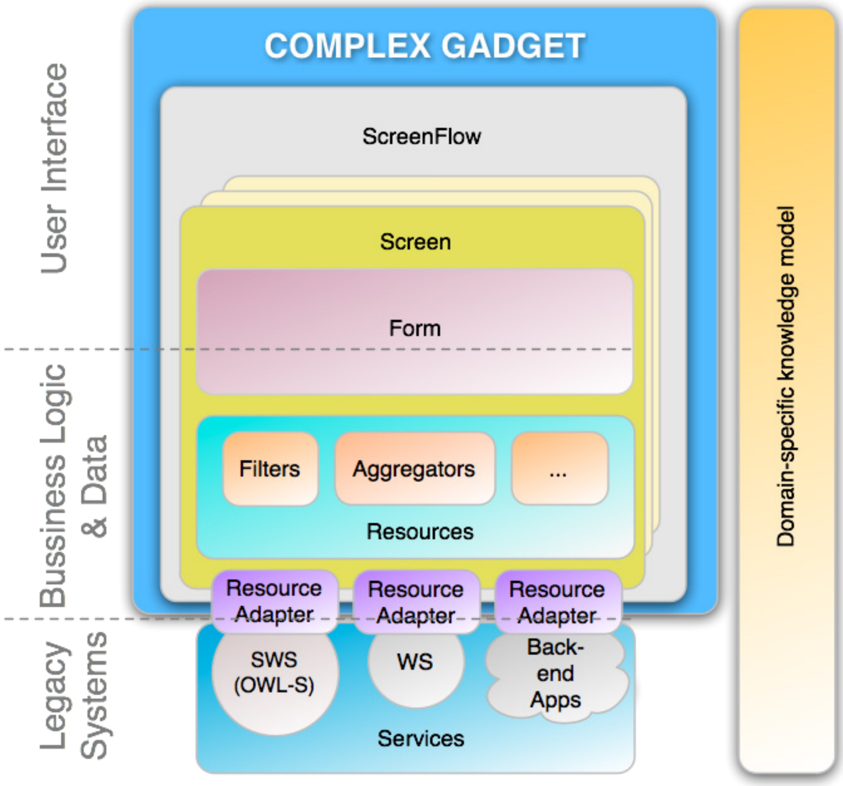


Figure 1: Complex Gadget Architecture

button to get a list of items for sale, or a checkout action performed in order to buy one of the items from the list. As an analogy, an action is what a operation is in a SOAP-based web service. Moreover, as these operations require some input, the actions may need a certain precondition (green circles) to be satisfied. And at last, the execution of these action may cause a postcondition (red circle) be satisfied, or following the same example, an output is returned as result of the execution of an operation.

3.2 Building

This section describes different approaches for describing and implementing web services, and how a resource adapter could be generated for these services.

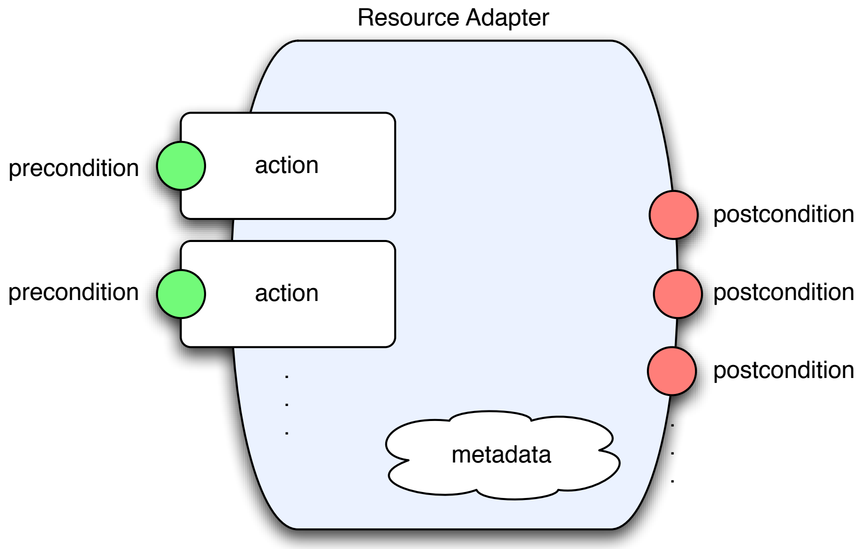


Figure 2: Resource Adapter Overview

3.2.1 REST-based Web Services

REST [Fielding, 2000] is a term used to describe an architecture style, not a standard, of networked systems. The acronym REST stands for Representational State Transfer. REST-based or RESTful web services are created identifying all the conceptual entities or resources which need to be exposed as services. Those entities or resources should be nouns, not verbs (orders, tickets, etc.). Then, the interaction with those resources is made by convention using HTTP verbs such as GET, POST, PUT or DELETE, in order to retrieve, create, modify or delete them.

RESTful web services are more light-weight than SOAP-based web services, avoiding the verbose XML-based messages, but unlike the former, which have a standard and commonly used vocabulary to describe the web service interface through WSDL, RESTful web services are currently not formally described most of the times. For a service consumer to understand the context and content of the data that must be sent to and received from the service, both the service consumer and service producer must have an out-of-band agreement. This takes the form of documentation, sample code, and an API that the service provider publishes for developers to use. For example, the most web-based services available from Google, Yahoo, Flickr, Amazon, and so on have accompanying artifacts describing how to consume the services. This style of documenting REST-based web services is fine for developers' use, but it averts tools from programmatically consuming such services and generating executable stubs for a given programming language, as a web service described using WSDL allows. Nevertheless, Web Application Description Language (WADL) attempts to resolve some of these issues by providing a means to describe services in terms of schemas, HTTP methods, and the request or response structures exchanged, but this language is not widely adopted yet by RESTful web service developers.

To permit a high number of REST-based web services to be integrated to the FAST platform, the approach taken is from a manual development perspective. There is no need of a formal document such as WADL defining the web service, for this reason, building service wrappers for these services involve the correct understanding of the service by a human being and a tool to facilitate this task. This tool is explained in detail in Section 4.

3.2.2 SOAP Web Services

SOAP-based web Services or "Heavyweight Web Services" use Extensible Markup Language (XML) [Bray et al., 2006] messages that follow the Simple Object Access Protocol (SOAP) standard [Group, 2003] and have been popular within traditional enterprise, usually relying on HTTP for message negotiation and transmission. In such services, there is often a machine-readable description of the operations offered by the service written in a Web Services Description Language (WSDL) document. The advantage of specifying the service using WSDL is that it can be programmatically processed.

Shortly, a WSDL definition of a service, regarding the WSDL 2.0 specification [Chinnici et al., 2007], will contain the following information:

Interfaces A set of Interface components describing sequences of messages that a service sends and/or receives.

Bindings A set of Binding components describing concrete message formats and transmission protocols which may be used to define the endpoints.

Services A set of Service components describing a set of endpoints at which a particular deployed implementation of the service is provided.

Element declarations A set of Element Declaration components defining the name and content model of the element information items such as that defined by an XML Schema global element declaration.

Type definitions A set of Type Definition components defining the content model of the element information items such as that defined by an XML Schema global type definition.

From a specific WSDL definition, a set of methods or operations can be easily extracted which encloses a set of inputs and outputs. These operations would be transformed into *actions*, and the inputs and outputs would be used to define resource adapter *pre/postconditions*. But there is a critical problem following this approach. The definition of the inputs, outputs, operations and their types are done in a XML Schema document. An XML schema describes, in a syntactic manner, the structure of an XML document, but there is no straight-forward relation with the concepts used to describe pre and postconditions in the building blocks. In order to illustrate the concepts of WSDL, and the gap between them and semantically-defined building blocks, a shorten version of a real Amazon E-commerce web service will be presented. Concretely, the choosen web service performs an item search through the Amazon catalog. It expects as input an ItemSearch element, which may contain an actor, an artist or an author, and the item availability. The output of the service would be a set of items. The WSDL including for this service is given below.

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://webservices.amazon.com/AWSECommerceService/2009-11-01"
  targetNamespace="http://webservices.amazon.com/AWSECommerceService/2009-11-01">

  <types>
   <xs:schema
      targetNamespace="http://webservices.amazon.com/AWSECommerceService/2009-11-01"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://webservices.amazon.com/AWSECommerceService/2009-11-01"
      elementFormDefault="qualified">

      <xs:element name="ItemSearch">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Author" type="xs:string" minOccurs="0"/>
            <xs:element name="Title" type="xs:string" minOccurs="0"/>
            <xs:element ref="tns:AudienceRating" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="MinimumPrice" type="xs:nonNegativeInteger" minOccurs="0"/>
            <xs:element name="MaximumPrice" type="xs:nonNegativeInteger" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="ItemSearchResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="tns:Items" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
  <message name="ItemSearchRequestMsg">
    <part name="body" element="tns:ItemSearch"/>
  </message>
  <message name="ItemSearchResponseMsg">
    <part name="body" element="tns:ItemSearchResponse"/>
  </message>

  <portType name="AWSECommerceServicePortType">
    <operation name="Help">
      <input message="tns:HelpRequestMsg"/>
      <output message="tns:HelpResponseMsg"/>
    </operation>
    <operation name="ItemSearch">
      <input message="tns:ItemSearchRequestMsg"/>
      <output message="tns:ItemSearchResponseMsg"/>
    </operation>
  </portType>

  <binding name="AWSECommerceServiceBinding" type="tns:AWSECommerceServicePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="ItemSearch">
      <soap:operation soapAction="http://soap.amazon.com/ItemSearch"/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
    </operation>
  </binding>

  <service name="AWSECommerceService">
    <port name="AWSECommerceServicePort" binding="tns:AWSECommerceServiceBinding">
      <soap:address
       location="https://ecs.amazonaws.com/onca/soap?Service=AWSECommerceService"/>
    </port>
  </service>

</definitions>
```

As stated before, one of the advantages of describing web services in a formal document is that they can be programmatically consumed. In fact, there are many frameworks out there to facilitate such a task, parsing WSDL files, creating stubs to consume the web service and so on. However, in order to create a resource adapter from the WSDL defined above, the types defined in the XML Schema need to be translated into concepts from an ontology.

This problem could be addressed in two different ways. First one, as a manual process mapping these types and their attributes to concepts and properties, as it was proposed for RESTful services. Second approach is taking advantage of SAWSDL, which would help ensure a progra-matical (mapping) process.

SAWSDL [Farrell and Lausen, 2007] stands for Semantic Annotations for WSDL and XML Schema, and it defines how to add semantic annotations to various parts of a WSDL document such as input and output message structures, interfaces and operations. These semantic annotations enable the possibility to develop resource adapters from WSDL documents. They reference a concept in an ontology or a mapping document, and are either independent of the ontology expression languages or mapping languages.

Basically, there are three main attributes used in SAWSDL: *modelReference* to specify the association between a WSDL or XML Schema component and a concept from the ontology, and *liftingSchemaMapping* and *loweringSchemaMapping* to define the mapping mechanisms between semantic data and XML. These lowering and lifting schema mappings are not explained in this deliverable, however in Appendix A of the SAWSDL specification [Farrell and Lausen, 2007] they are described in detail, and illustrated with examples.

To illustrate the usage of these attributes, the following code shows a fragment of the previous example using SAWSDL to semantically annotate the service with concepts from an imaginary ontology defining the Amazon E-commerce web service.

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
  xmlns:tns="http://webservices.amazon.com/AWSECommerceService/2009-11-01"
  targetNamespace="http://webservices.amazon.com/AWSECommerceService/2009-11-01">

  <types>
   <xs:schema
      targetNamespace="http://webservices.amazon.com/AWSECommerceService/2009-11-01"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://webservices.amazon.com/AWSECommerceService/2009-11-01"
      elementFormDefault="qualified">

      <xs:element name="ItemSearch"
       sawsdl:modelReference="http://ontologies.amazon.com/AWS#ItemSearch"
       sawsdl:loweringSchemaMapping="http://mapping.amazon.com/AWS/Ont2ItemSearch.xml"
       sawsdl:liftingSchemaMapping="http://mapping.amazon.com/AWS/ItemSearch2Ont.xslt">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Author" type="xs:string" minOccurs="0"/
             sawsdl:modelReference="someURI"
             sawsdl...>
            <xs:element name="Title" type="xs:string" minOccurs="0"/>
            <xs:element ref="tns:AudienceRating" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="MinimumPrice" type="xs:nonNegativeInteger" minOccurs="0"/>
            <xs:element name="MaximumPrice" type="xs:nonNegativeInteger" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="ItemSearchResponse"
       sawsdl:modelReference="http://ontologies.amazon.com/AWS#ItemSearch"
       sawsdl:loweringSchemaMapping="http://mapping.amazon.com/AWS/Ont2ItemSearch.xml"
      sawsdl:liftingSchemaMapping="http://mapping.amazon.com/AWS/ItemSearch2Ont.xslt">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="tns:Items" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>

  <message name="ItemSearchRequestMsg"
   sawsdl:modelReference="http://ontologies.amazon.com/AWS#ItemSearchRequest">
    <part name="body" element="tns:ItemSearch"/>
  </message>
  <message name="ItemSearchResponseMsg"
   sawsdl:modelReference="http://ontologies.amazon.com/AWS#ItemSearchResponse">
    <part name="body" element="tns:ItemSearchResponse"/>
  </message>

  ...

</definitions>
```

Hence, SAWSDL enriches the web services descriptions made in WSDL with references to concepts from a semantic model and enables the possibility to create resource adapters from these definitions without a strong user participation in the process. If the semantic metadata is missing, a manual annotation process could be considered, where the user would need to relate concepts from an ontology with the WSDL or XML Schema elements.

3.2.3 WSMO Web Services

Semantic web services bring a number of advantages in the creation of the resources adapters over classic web services. Formal and semantic descriptions of web services allow the creation of mechanisms to raise their exploitation in a more automatic way. In this section, some general notions of the Web Service Modelling Ontology (WSMO) will be explained and how FAST can make use of it.

In a few words, WSMO provides means to describe all relevant aspects of semantic web services in a unified manner. A web service description in WSMO consists of five sub-components: non-functional properties, imported ontologies, used mediators, a capability and interfaces. However, we will focus on the capabilities and the interfaces since they are components which will make possible the integration of these services in FAST.

Capabilities and interfaces are the two types of web service description in WSMO. The capabilities describe the different functions of the web service, while the interfaces specify:

1. How to communicate with a web service in order to avail of its functionality. This is called Choreography.

2. How the functionality of a web service is enabled by interacting with other web services. This is called Orchestration.

A web service in WSMO defines one and only one capability. The capability of a web service defines its functionality in terms of pre/postconditions, assumptions and effects. Apart from that, a web service capability is defined by specifying the following elements: non-functional properties, imported ontologies, used mediators, and shared variables. The creation of the resource adapters for these web services will just take into consideration the set of pre/postconditions.

In the rest of the section, the Amazon E-commerce web service will be used to build a WSMO web service with the same functionality [Kopecky et al., 2005]. In fact, just the small subset of the service, used in previous sections, will be tranformed.

First of all, an ontology is needed describing the Amazon domain. For the sake of simplicity, it will be created doing a one-to-one mapping between the XML Schema types and the concepts, though a real sceneario may consider a different approach to optimise the modelling.

The concept describing an item search request for the Amazon web service is:

```
concept itemSearchRequest
    nonFunctionalProperties
      dc#description hasValue "A request to search for an item"
    endNonFunctionalProperties
    author ofType _string
    title ofType (1) _string
    rating ofType ratings#audienceRating
    minPrice ofType (1) _decimal
    maxPrice ofType (1) _decimal
```

The concept *itemContainer* describes a list of items returned in the item search response, and its code is presented below:

```
concept itemContainer
    nonFunctionalProperties
      dc#description hasValue "Contains a list of items"
    endNonFunctionalProperties
    items ofType item
```

Next step would be to describe the namespace and non-functional properties declarations for the WSMO Amazon web service, but as it was said, this information is not relevant for the example. Hence, the simplified version of the Amazon web service preconditions would look like:

```
precondition
  nonFunctionalProperties
     dc#description hasValue "The Amazon service handles requests
     for searching for items by keywords"
  endNonFunctionalProperties
  definedBy
    // A search by author, title, minPrice or maxPrice
    ?request memberOf am#itemSearchRequest
```

It is a fairly simple precondition, for which the only condition required is the existence of a *itemSearchRequest* object.

However, postconditions are a bit more difficult to construct. They take the form of implication rules, so given a particular precondition, one particular postcondition will take effect. To follow the same example, these would be the postconditions for the Amazon search items functionality, including just the criterion author, title, minimum and maximum price as defined in the ontology.

```
 postcondition
  nonFunctionalProperties
    dc#description hasValue "The service returns a list of searched
      items"
  endNonFunctionalProperties
  // The result for a search request by author is a container of products
  // such that all items in the container have the requested author
  (?request[
      author hasValue ?author
    ] memberOf am#searchItems implies
    exists ?container
      (?container memberOf am#itemContainer and
        forall ?item
          (?container[
              items hasValue ?item
            ] implies
            ?item[
              author hasValue ?author
            ]
          )
      )
  ) and

  // The result for a search request by title is a container of products
  // such that all items in the container have the requested title
  (?request[
      title hasValue ?title
    ] memberOf am#searchItems implies
    exists ?container
      (?container memberOf am#itemContainer and
        forall ?item
          (?container[
              items hasValue ?item
            ] implies
            ?item[
              title hasValue ?title
            ]
          )
      )
  ) and

  // The result for a search request by minPrice is a container of products
  // such that all items in the container have a price >= requested minPrice
  (?request[
      minPrice hasValue ?minPrice
    ] memberOf am#searchItems implies
    exists ?container
      (?container memberOf am#itemContainer and
      forall{?item,?price}
        (?container[
            items hasValue ?item
          ] and
          ?item[
            price hasValue ?price
          ] implies
          ?price >= ?minPrice
        )
      )
  ) and

  // The result for a search request by maxPrice is a container of products
  // such that all items in the container have a price <= requested maxPrice
  (?request[
      maxPrice hasValue ?maxPrice
    ] memberOf am#searchItems implies
    exists ?container
      (?container memberOf am#itemContainer and
      forall {?item,?price}
        (?container[
            items hasValue ?item
          ] and
          ?item[
            price hasValue ?price
          ] implies
          ?price =< ?maxPrice
        )
      )
  ).
```

The manner pre/postconditions are defined in WSMO web services are similar to the way they are defined in the Resource Adapters. Section 4.4 in [Möller and Rivera, 2010] gives a detailed explanation on this topic. Basically, using previous examples, the preconditions would be transformed as follows:

```
?request memberOf am#itemSearchRequest => ?request rdf:type am:itemSearchRequest
```

And a simple postcondition would be transformed as follows:

```
exists ?container

      (?container memberOf am#itemContainer) => ?container rdf:type am:itemContainer
```

In the previous example, there is not a formal expression saying that *?container* must exists, but it is assumed.

3.3 Discovery

In the building phase, every resource adapter is created sharing a common structure for screen components, such as forms, operators and resources. Hence, it will have a set of actions which will contain a set of preconditions to be satisfied in order to be executed, and after its execution, it may produce any of the conditions of their postconditions.

For this reason, a screen developer during the development phase is able to reuse these screen components which are already stored in the catalogue, establishing which the screen's pre and postconditions are, and then the system will suggest several screen components which may satisfy these pre or postconditions. In a first step, the system will suggest screen components having any of the required preconditions, which can be connected to any of the screen preconditions (the connection will be represented by an internal pipe), and the same would happen for the postconditions. Once a screen component is inserted into the screen, the system will take its definition into consideration to recommend new screen components which can be connected to the ones the screen is composed of.

On these basis, the mechanism is relatively similar to the screen recommender, used to find the best screens to create a screen flow. Once more, it is based on the pre/postconditions of a set of building blocks.

3.4 Connection

The aim of a resource adapter is to perform some action on a web service. In order to perform a given action, a precondition may need to be fulfilled, and as a result of the execution a postcondition may be generated. These pre/postconditions are intended to be used by pre/postconditions of other building blocks within a gadget. This connection must be explicitly defined, because it will indicate the real data-flow between different building blocks. In FAST this is commonly called Piping, since a pipe in computer science is just that, a one-way communication channel for interprocess communication; therefore a pipe is a logical connection made between a precondition and a postcondition of different building blocks. Figure 3 shows an example of a search form for the Amazon service. This is a screen, composed by a form and a resource adapter. The form allows the user to introduce a search term, producing a postcondition which will be redirected to a resource adapter precondition. The execution of the web service produces a list of results, which will be forwarded to the postcondition of the screen.

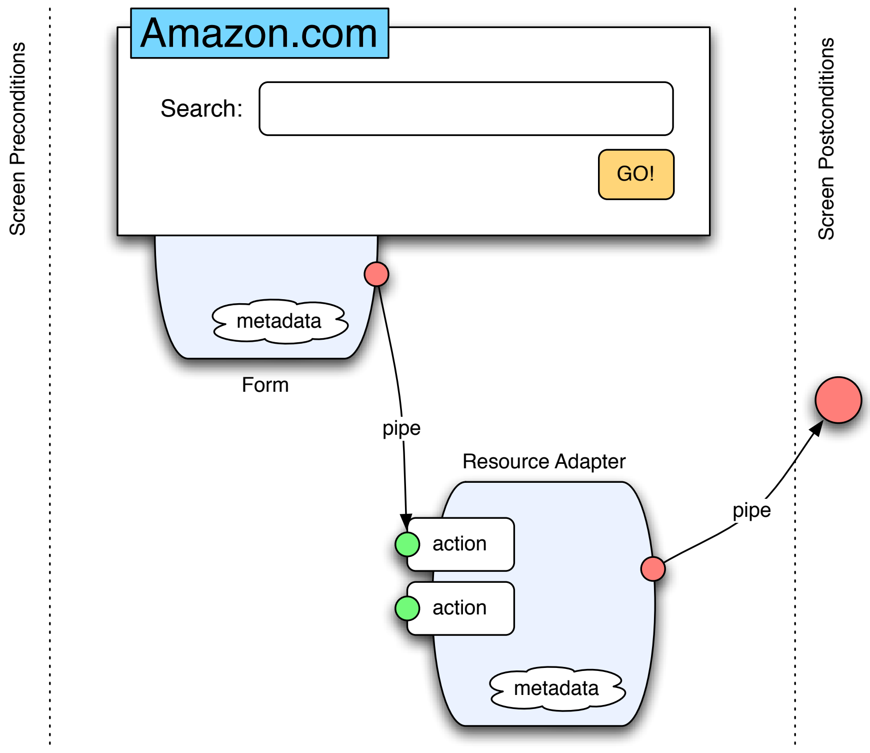


Figure 3: Piping

4 RESTful Web Services Wrapper Tool

In FAST, the wrapper tool will be in charge of the resource adapters' building phase. At the moment, it just supports RESTful web services. The building phase for this type of web services wrapper is done in two steps: a first step will be in charge of the construction of a service request and a second step will analyse the response got from the execution of the service, allowing the extraction of facts contained in that response and mapping them to domain-specific concepts from the ontologies used within FAST by any building block.

4.1 Constructing service requests

As a first approach, the interaction with these services will be limited to retrieve information to feed the gadgets using simply GET requests. A service request would be assembled using a certain URL and a set of parameters. As an example, Ebay Shopping web service will be studied. The following URL is invoked to retrieve a list of items corresponding to certain search keywords:

```
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.

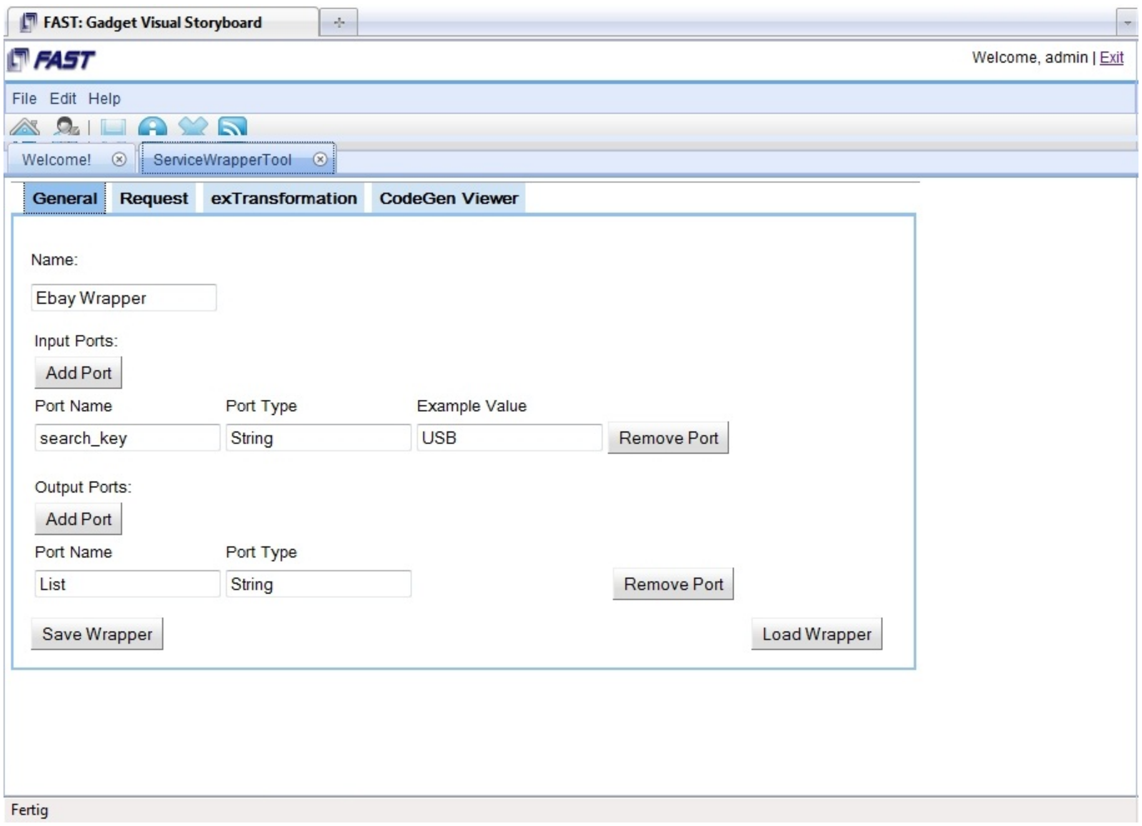
responseencoding format of the response message obtained by the invocation of the request.

If needed, a detailed specification of the Ebay Shopping API can be found at [eBay, 2010].

The above URL for searching items in Ebay is followed by the query parameters, which take the form *argument=value*, where the arguments and values are URL encoded, and are separated by an ampersand (&). For instance, the only relevant parameter the user would need to specify is *querykeywords*, thus somehow the service has to receive an input value for it, while the other parameters can be set to a default value. However, in order to develop a generic wrapper for the any service, all parameters might be set by the user.

To achieve this, service wrappers will be handled as any other building block with preconditions and postconditions (inputs and outputs). Therefore, these precondition ports might be used to determine values for parameters like query keywords, coming from a building block inside the screen in which the service is placed or even an external screen, as a screen precondition.

To define the preconditions and postconditions of a new service wrapper, the FAST service wrapper tool provides a form allowing the edition of these entries, cf. Figure 4[2]



Name: Ebay Wrapper

Input Ports:

Port Name	Port Type	Example Value
search_key	String	USB

Output Ports:

Port Name	Port Type
List	String

Figure 4: Configuring pre/postconditions of a service wrapper

The service wrapper tool composes the request using a template string which will contain placeholders for precondition values. An example of a template with one placeholder (<search_query>), to perform a search on google could be like:

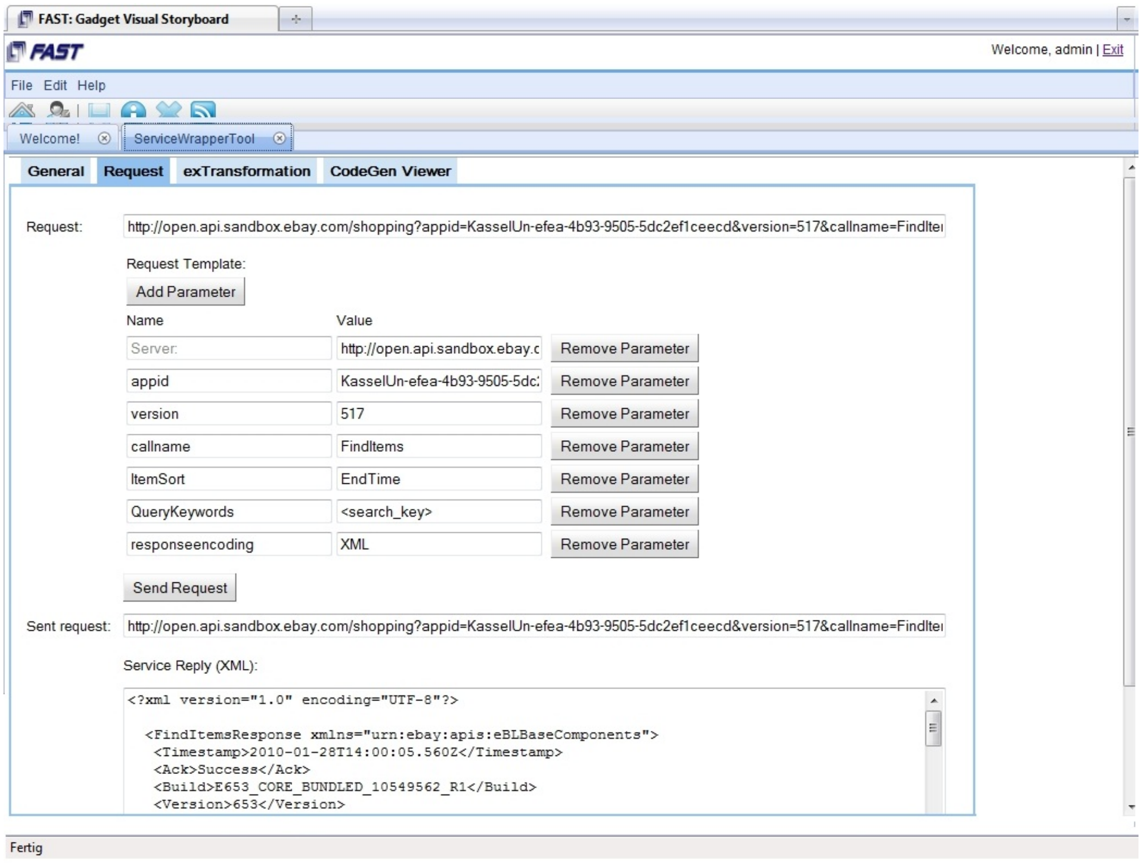
```
http://www.google.com/search?lang=es&q=<search_query>
```

[2]Note: These example values may be used to test the service wrapper, but they can be freely edited.

Before sending the request to the service, the placeholders are filled with their corresponding values and the resulting URL is then ready to be sent as the service request. Following the previous example, the placeholder *search_query* is filled with a specific search term:

```
http://www.google.com/search?lang=es&q=six+nations+2010
```

Figure 5 shows the screen to construct the service requests. In the top input field of Figure 5, the user may drop an example HTTP request taken e.g. form the service documentation (e.g. [eBay, 2010] for this example). The tool analyses the example request and, in the middle of the screen, a form for editing the request parameters is provided. This example shows how a user has connected the *QueryKeywords* parameter with the precondition *search_key* by adding a corresponding reference to the value field of that parameter. In addition, an access key has been retrieved and has been entered as value for the *appid* parameter.



Request: http://open.api.sandbox.ebay.com/shopping?appid=KasselUn-efea-4b93-9505-5dc2ef1ceecd&version=517&callname=FindIte

Request Template:

Name	Value
Server:	http://open.api.sandbox.ebay.c
appid	KasselUn-efea-4b93-9505-5dc
version	517
callname	FindItems
ItemSort	EndTime
QueryKeywords	<search_key>
responseencoding	XML

Sent request: http://open.api.sandbox.ebay.com/shopping?appid=KasselUn-efea-4b93-9505-5dc2ef1ceecd&version=517&callname=FindIte

Service Reply (XML):

```
<?xml version="1.0" encoding="UTF-8"?>

  <FindItemsResponse xmlns="urn:ebay:apis:eBLBaseComponents">
   <Timestamp>2010-01-28T14:00:05.560Z</Timestamp>
   <Ack>Success</Ack>
   <Build>E653_CORE_BUNDLED_10549562_R1</Build>
   <Version>653</Version>
```

Figure 5: Constructing the service request URL and its parameters

Below the request parameter editing form of Figure 5, a *Send Request* button allows to validate the constructed service by sending it to the specified service address (via a server relay). Then the placeholders for preconditions are replaced with 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, which helps the users to figure out quickly whether the constructed request works as desired.

4.2 Interpreting service responses

Once the service request is constructed and sent to the service provider, it will send back a response. This response message could be serialized in any format, though the most common formats used nowadays are XML or JSON among others. To continue the example that was started in the previous section, the response of the Ebay Shopping service will be in XML as specified in the request, which is the format supported by the wrapper tool.

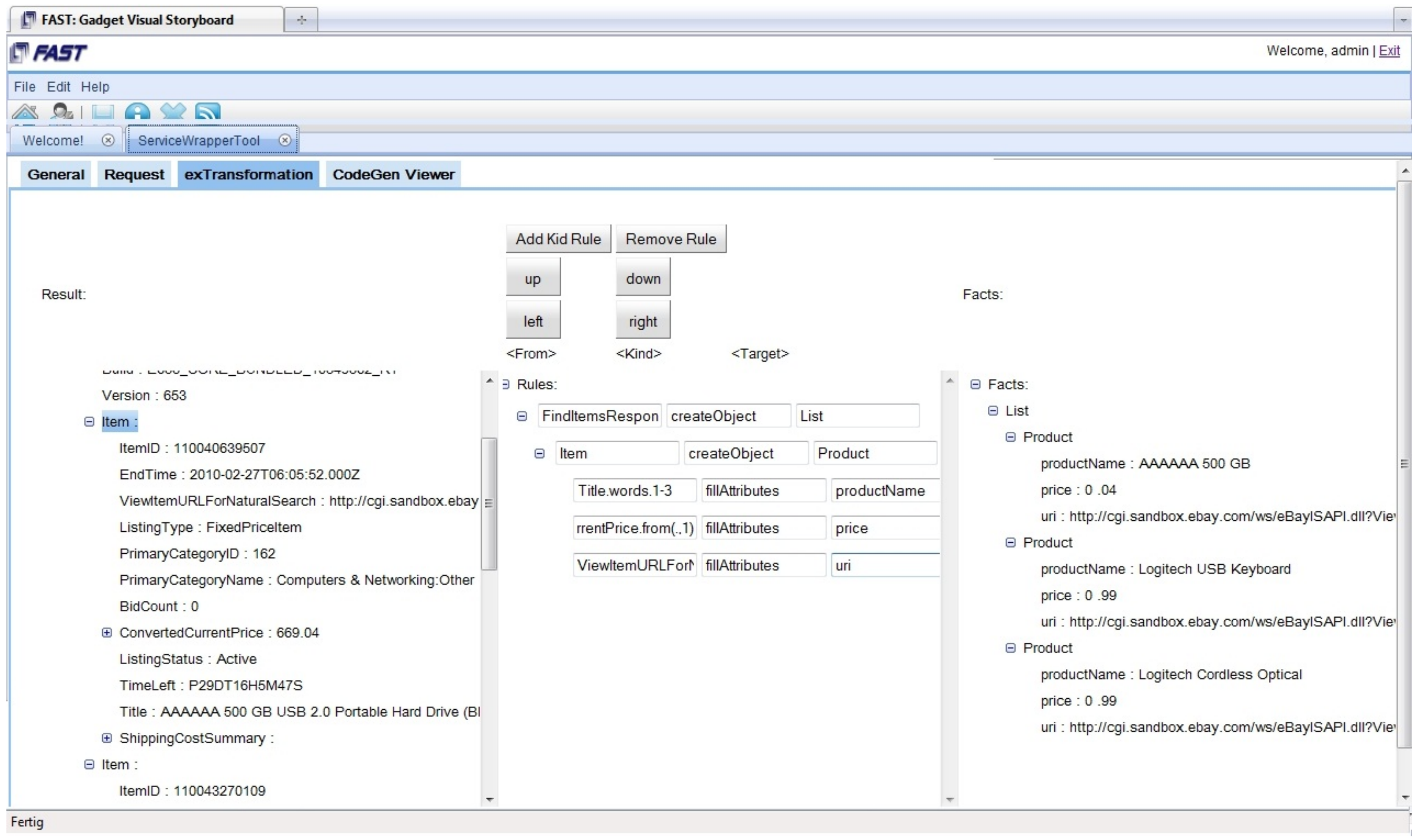
Figure 6 shows the transformation data tab of the wrapper tool.

Once the service response, in XML format, has been retrieved, the transformation tab shows it as an interactive object tree on the left side of Figure 6. To construct this interactive object, the XML document has been parsed into a DOM, and a simplified tree representation of that DOM is built up. This tree representation of the XML data is used as an input to construct transformation rules.

The rule based approach of the service wrapper tool has been inspired by triple graph grammars [Schuerr, 1994, Jahnke et al., 1997]. While general triple graph grammars allow to relate complex structures to each other, in the approach taken for FAST, the target of a rule is always a single object or attribute making the process as easy as possible for the user.

A transformation rule is used to analyse the XML data and to generate domain-specific facts from concepts from the ontologies used by the pre/postconditions of the different building blocks. A transformation rule is composed of three elements (see the middle part of Figure 6). First, the *from* field indicates the XML elements to be translated by rule. These XML elements are identified by the tagname 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 triggered from the transformation rules is:

action *createObject* specifies the creation of a new fact object. The type of that new fact is provided in the third compartment. In the example being explained, the root rule searches



Result:

```
Version : 653
⊟ Item :
    ItemID : 110040639507
    EndTime : 2010-02-27T06:05:52.000Z
    ViewItemURLForNaturalSearch : http://cgi.sandbox.ebay
    ListingType : FixedPriceItem
    PrimaryCategoryID : 162
    PrimaryCategoryName : Computers & Networking:Other
    BidCount : 0
  ⊞ ConvertedCurrentPrice : 669.04
    ListingStatus : Active
    TimeLeft : P29DT16H5M47S
    Title : AAAAAA 500 GB USB 2.0 Portable Hard Drive (Bl
  ⊞ ShippingCostSummary :
⊟ Item :
    ItemID : 110043270109
```

```
<From>            <Kind>            <Target>
⊟ Rules:
 ⊟ FindItemsRespon   createObject    List
   ⊟ Item            createObject    Product
       Title.words.1-3     fillAttributes   productName
       rrentPrice.from(.,1) fillAttributes  price
       ViewItemURLForN     fillAttributes   uri
```

Facts:

```
⊟ Facts:
 ⊟ List
   ⊟ Product
       productName : AAAAAA 500 GB
       price : 0 .04
       uri : http://cgi.sandbox.ebay.com/ws/eBayISAPI.dll?Vie
   ⊟ Product
       productName : Logitech USB Keyboard
       price : 0 .99
       uri : http://cgi.sandbox.ebay.com/ws/eBayISAPI.dll?Vie
   ⊟ Product
       productName : Logitech Cordless Optical
       price : 0 .99
       uri : http://cgi.sandbox.ebay.com/ws/eBayISAPI.dll?Vie
```

Figure 6: Interactive rule based transformation of an XML response to FAST facts

for XML elements with tagname *FindItemsResponse* and for each such element a *List* fact is created. The resulting facts are shown in a facts tree in the right of Figure 6.

action *fillAttributes* does not create a new fact 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 tagname *Title*. Note, the rule is a sub-rule of the second rule, which generates *Product* facts. 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 6). The value of that *Title* element is then transfered to the *productName* attribute of the corresponding *Product* fact. Actually, our *from* fields allow 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 facts but such rules are just used to narrow the search space for their sub-rules. For example, in the Amazon case, 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.

Since FAST is storyboard oriented, the service wrapper tool follows the storyboard idea as well. Any time, a change to a transformation rule is done, the transformation process is triggered and the resulted 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, FAST is semantic-driven, therefore, the service wrapper designer shall retrieve the domain-specific types from a FAST ontology server together with the structure of each type, i.e. together with a description of the attributes of each fact. 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 fact types available for that domain. For the *fillAttributes* rules, the selection box shows the attributes of the fact type chosen in the parent rule. In addition, we may provide some analysis tool, which will help to guarantee that the facts generated by the transformation rules conform to the fact types defined in the corresponding FAST ontology. This helps to ensure that the facts generated will be compatible for precondition ports of subsequent filter steps.

4.3 Generating a Resource Adapter

Once the wrapping of a service has been defined and tested in the service wrapper tool, it shall generate an implementation of the desired Resource Adapter in JSON, HTML, and JavaScript, ready to be deployed and executed inside a screen. This service wrapper implementation is compliant with the formats required by the Gadget Visual Storyboard Tool (GVS) and it shall be stored inside the FAST catalogue, in order to be found and used by any user.

4.4 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 most 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 tranforamtion 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 fact formats can be used which would allow generating facts to be further processed by additional filters. Such additional filters may be realized using (hand coded) operators, since some generic operators can act as filters for aggregation and conversions of facts from multiple sources. Then, service wrappers in combination with these filter operators will allow covering some of these complex cases.

References

[Alonso et al., 2003] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2003). *Web Services*. Springer.

[Bray et al., 2006] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowa, J. (2006). Extensible markup language (xml) 1.1 (second edition).

[Chinnici et al., 2007] Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. (2007). Web services description language (wsdl) version 2.0.

[eBay, 2010] eBay (2010). ebay shopping apis.

[Farrell and Lausen, 2007] Farrell, J. and Lausen, H. (2007). Semantic annotations for wsdl and xml schema. w3c recommendation.

[Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

[Gordon and Sensen, 2008] Gordon, P. and Sensen, C. (2008). Creating bioinformatics semantic web services from existing web services: A real-world application of sawsdl. In *Proceedings of the IEEE International Conference on Web Services, Beijing, China*.

[Group, 2003] Group, X. P. (2003). Soap version 1.2. W3c recommendation, W3C.

[Jahnke et al., 1997] Jahnke, J.-H., Schaefer, W., and Zuendorf, A. (1997). A Design Environment for Migrating Relational to Object Oriented Database Systems. *Software Engineering and Database Technology (Dagstuhl-Seminar-Report 173)*.

[Kopecky et al., 2005] Kopecky, J., Roman, D., and Scicluna, J. (2005). D3.4v0.2. wsmo use case: Amazon e-commerce service. Technical report, DERI.

[Möller and Rivera, 2010] Möller, K. and Rivera, I. (2010). Ontology and conceptual model for the semantic characterisation of complex gadgets. Technical report, FAST Project (FP7-ICT-2007-1-216048).

[Schuerr, 1994] Schuerr, A. (1994). Specification of graph translators with triple graph grammars. In Mayr, E. W., Schmidt, G., and Tinhofer, G., editors, *WG*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer.

[Urmetzer et al., 2010] Urmetzer, F., Delchev, I., Hoyer, V., Janner, T., Rivera, I., Möller, K., Aschenbrenner, N., Fradinho, M., and Lizcano, D. (2010). State of the art in gadgets, semantics, visual design, SWS and catalogs. Deliverable D2.1.2, FAST Project (FP7-ICT-2007-1-216048).