FAST AND ADVANCED STORYBOARD TOOLS

FP7-ICT-2007-1-216048

http://fast.morfeo-project.eu

Deliverable D4.3.2

Mechanisms for Gadget-Service Connections and Gadget Functionality

Ismael Rivera (NUIG)
Albert Zündorf (UniKas)
Knud Möller (NUIG)

Date: 27/02/2010

Version History

Rev. No.	Date	Author (Partner)	Change description
1.0	26.02.2010	Ismael Rivera (NUIG), Albert Zündorf (UniKas)	Final version ready for external review
2.0	27.02.2011	Ismael Rivera (NUIG), Albert Zündorf (UniKas) Knud Möller (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 in what in the FAST platform is called Resource Adapters. The application of semantics to the back-end, through the corresponding Resource Adapters, and front-end building blocks assures 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 exploiting 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**	D4.3.2	**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**	28/02/2011	**Submitted**	27/02/2011
Status			final	
Dissemination Level	Public ⊠ / Consortium ☐			
Short description of contents	D4.3.2 is...			
Authors	Ismael Rivera (NUIG), Albert Zündorf (UniKas), Knud Möller (NUIG)			
Deliverable Owner	Ismael Rivera (NUIG)		**email**	ismael.rivera@deri.org
(Partner)			**phone**	+353 91 ?????
Keywords	FAST, web services, WSDL, REST, SOAP, WSMO			

Table of contents

1 Introduction

1.1 Goal and Scope

The objective of this deliverable is the analysis and development of mechanisms to facilitate the connection between screen-flow gadgets and underlying Web services, based on front-end user requirements and on the semantic descriptions of the service wrappers, and to develop these service wrappers from of 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. Section 2 describes what a service wrapper is, how the platform is able to interact with web services, how these wrappers are build and then discovered to be reused in any gadget. Finally, Section 3 defines the service wrapper tool built to facilitate FAST users to create Resource Adapters of RESTful web services.

1.3 Changes from Previous Version

To be done...

2 Web Services Wrapping

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 to communicate the front-end (i.e. the gadget user interface and logic) 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 shift 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.

Therefore, 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 the discovery and use of them by other components (i.e. forms and operators). This is called Resource Adapter in the FAST platform. Figure 1 illustrates their position and interaction with the rest of the building blocks within a gadget.

Once a brief background has been shown, the following sections explain how a user, in this case a resource developer, can build Resource Adapters, and how another user, a screen developer for instance, will discover and connect them while creating a screen. Hence, these three phases are called: building, discovery and connection.

2.1 Building

2.1.1 REST-based Web Services

REST is a term 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 [Fielding, 2000] are created identifying all of the conceptual entities or resources which want to be exposed as services. Those entities or resources should be nouns, not verbs (orders, tickets,
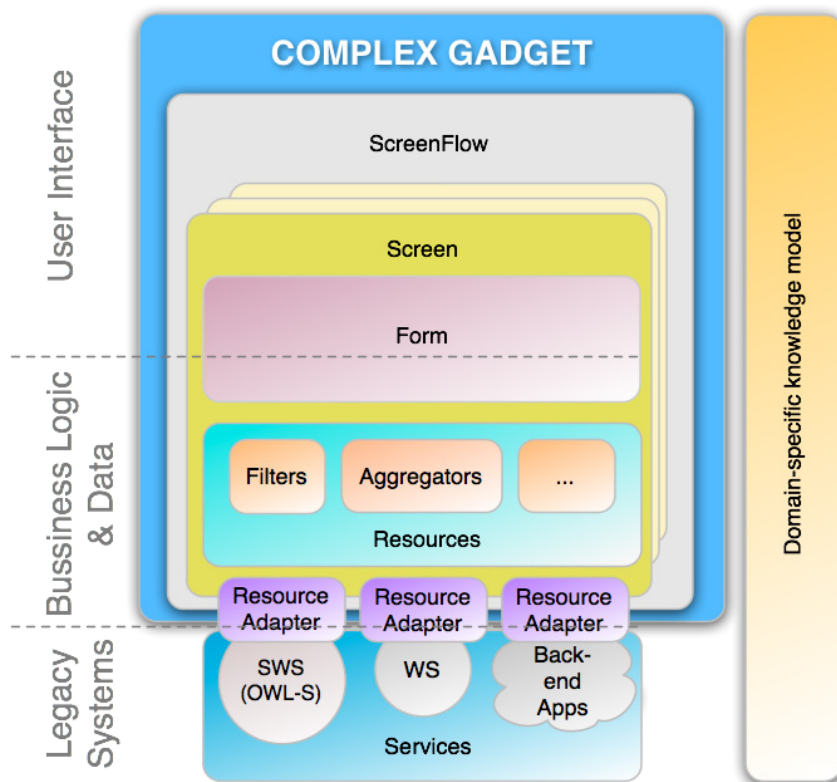
Figure 1: Complex Gadget Architecture

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.

REST is lightweight (not a lot of extra xml markup, human readable results), but unlike SOAP-based web services, which have a standard vocabulary, and commonly used, 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 many web-based services available from Google, Yahoo, Flickr, Amazon, and so on have accompanying artefacts describing how to consume the services. This style of documenting REST-based web services is fine for use by developers, but it averts tools from programmatically consuming such services and generating artefacts specific to programming languages, 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 3.

2.1.2   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 with 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. Hence, the advantage of 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-/post-conditions. With the use of this information along with the XML Schema defining the types, a Resource Adapter can be semantically defined.

To integrate WSDL defined services into the FAST platform, we again propose an interactivce approach: We extend the tool for REST service wrapping with the ability of loading WSDL definitions of a service and this WSDL definition is then exploited for guidance of the user through the Resource Adapter construction, cf. Section 3.

2.1.3 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 mechanisms

to raise its 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 WSMO, 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-/post-conditions, assumptions and effects. A web service capability is defined by specifying the following elements: non-functional properties, imported ontologies, used mediators, shared variables, pre-condition, post-condition, assumption, and effect.

Basically, the web service will offers to a client a post-condition when some conditions are met in the information space (pre-condition). Effects and assumptions can stay out of this study, since they will not be taken into account by any of the building blocks of a gadget.

Like WSDL defined services, we incorporate WSMO services into the FAST platform in an interactive storyboard driven approach. The FAST tool described in Section 3 reads the WSMO specification of a service and exploits this information again for guidance for the construction of a Resource Adapter.

2.2 Discovering

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 pre-conditions to satisfy in order to be executed, and after its execution, it may produce any of the conditions of their post-conditions.

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

For a further explanation on how the recommendation algorithms are implemented, please see [Urmetzer et al., 2010].
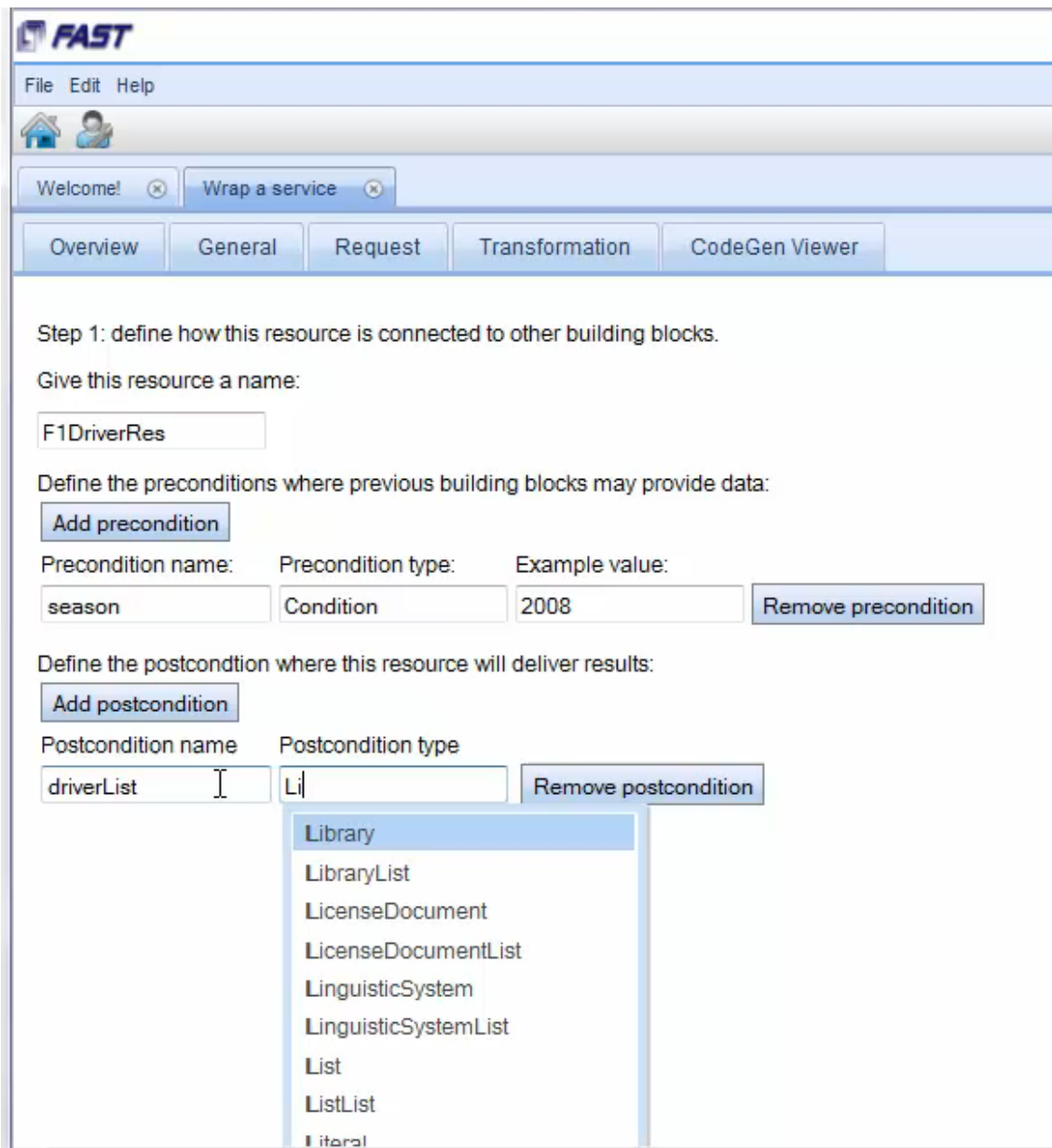
3 Service Wrapper Tool

In FAST, the Service Wrapper Tool will be in charge of the Resource Adapters' building phase. The building of resource adapters is done in three steps: in a first step, we define which pre- and post-conditions are used to pass parameters between FAST building blocks and the resource adapter. The second step is in charge of the construction of a service request and the third step analyses the response got from the execution of the service, allowing the extraction of facts contained in that response and mapping them to domain-specific classes or concepts from the ontologies used within FAST by the building blocks' conditions.

3.1 Defining pre- and post-conditions

We will demonstrate the use of the Service Wrapper Tool on the example of a simple *Formula1* service. In the first tab of the Service Wrapper Tool, the user defines the name and the pre- and post-conditions of the desired Resource Adapter, cf. Fig. 2. For the pre-condition, we provide a name, a type, and an example value. The type of a pre- or post-condition is chosen from a drop down list of available concepts. The list of available concepts, together with its properties and some example data (instances) is retrieved from the Catalogue. The example value is used for test requests during subsequent construction steps. We keep track of example values provided for different domain concepts and propose them, when the domain concept is re-used e.g. in another Resource Adapter.

3.2 Constructing service requests

In our simple case, the service request consists just of a URL that is easily retrieved from API documentation page of our Formula1 service. This URL consists of the service address `http://ergast.com/api` followed by the year we are querying for followed by the category `drivers` we are interested in. The user may grab such an URL from the API documentation page of the targeted web service and drop it into the Request tab of the Service Wrapper Tool, cf. Figure 3.

Figure 2: Configuring pre- and post-conditions of a Resource Adapter

The Service Wrapper Tool will than analyze the dropped URL and split it into its parts and provide a convenient form allowing to edit e.g. all the parameters of the URL. In our example the URL has no parameters so the parameter form shows only the base URL.

For WSDL or WSMO services, the Service Wrapper Tool provides additional support for the construction of the request. The user may point to the WSDL or WSMO definition of the targeted service. The Service Wrapper Tool then reads this definition and analyses it. With the help of this information, the Service Wrapper Tool provides interactive guidance for selecting the desired operation from a service and then it retrieves the operation descriptions and fills the request parameter form adequately.

Now we have to connect the request parameters to the parameter values passed via the preconditions to the Resource Adapter. In our case, we replace the year part of the URL with a placeholder for the pre-condition season `<season>`, see Request URL input field at the top of Figure 3.

In our example, we use a simple HTTP GET request. The Service Wrapper Tool also supports the other kinds of HTTP methods as POST, PUT or DELETE. For POST requests, one usually passes parameters within a special post body. For unstructured POST bodies, we provide a text area allowing the user to fill in placeholders for references to pre-condition values. In case of structured POST bodies as in the case of WSDL and WSMO services the POST body uses a structured text format based e.g. on JSON or XML. In that case, the body text is again passed and the parameters are included in the request parameter form part. Then the user may easily use the request parameter form part to fill in pre-condition placeholders.

WSDL is usually used with SOAP and XML, JSON wouldn't be common in this case. And from my understanding, the user has to hand-write the POST body? therefore the SOAP message? that's doable, but pretty hard and tedious. - Ismael

Is PUT and DELETE supported? if so, how? - Ismael

Figure 3 shows the screen to construct the service requests.

Below the request parameter editing form of Figure 3, a *Send Request* button allows to validate the constructed request by sending it to the specified service address (via a server relay). Then the placeholders for pre-conditions are replaced with example values and the resulting 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.

Figure 3: Constructing the service request and its parameters

3.2.1 Limitations

It is worth pointing out that currently the Service Wrapper Tool is able to construct input ports for the wrappers using just basic types. However, every building block accepts any (complex) concept as a pre-/post-condition, so the wrapper tool needs to be adapted to permit any concept as input or pre-condition, extracting the required text to construct the request. This should be easy to achieve before the project review.

I guess this paragraph will be rewritten...because my understanding is that we do support constructing wrappers with "complex" pre-/post-conditions and not just basic types. - Ismael

3.3 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 serialize in any format, though the most common formats used nowadays are XML or JSON. To continue the example started in the previous section, the response of the Formula1 service will is formatted in XML. Th Service Wrapper Tool supports both JSON and XML.

3.3.1 Translation XML/JSON into Facts

Figure 4 shows the data tranformation tab of the wrapper tool.

Once the service response has been retrieved, it is shown in the transformation tab as an interactive object tree, as it can be seeing on the left side of Figure 4. To construct this interactive object tree, the XML or JSON document is parsed into a DOM, and a simplified tree representation of that DOM is built up. This tree representation of the XML/JSON data is used as an input to construct transformation rules.

A transformation rule is used to analyse the XML/JSON data and to generate domain-specific facts from concepts from the ontologies used by the pre-/post-conditions of the different building blocks. A transformation rule is composed of three elements, as the middle part of Figure 4 shows.
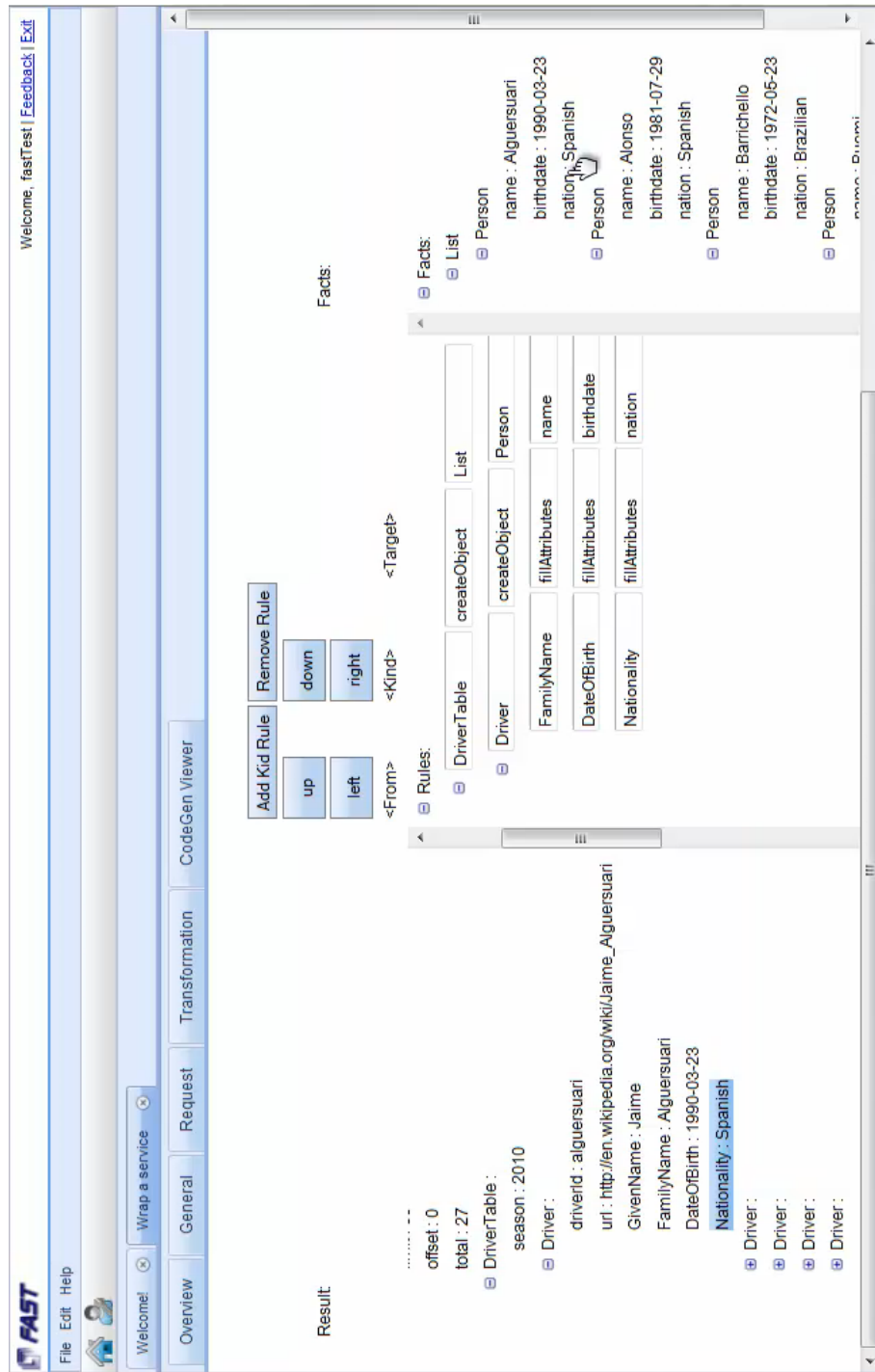
Figure 4: Interactive, rule based transforming of an XML response to FAST facts

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 for XML elements with tagname *DriverTable* and for each such element a *List* fact is created. The resulting facts are shown in a facts tree in the right of Figure 4.

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 *FamilyName*. Note, the rule is a sub-rule of the second rule, which generates *Person* facts. Thus, the sub-rule searches for *FamilyName* tags only in the sub-tree of the XML data that has been identified by an application of the parent rule before. For example, the *Driver* rule may just have been applied to the first *Driver* element of the XML data. Then, the *FamilyName* rule is applied only to the first *Driver* sub-tree of the XML data and thus it will find only one *FamilyName* element in that sub-tree. The value of that *FamilyName* element is then transfered to the *name* attribute of the corresponding *Person* fact. 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. A *from* field may also refer to multiple tags of the source tree, e.g. to combine a full name from a first name and a family name field.

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 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, FAST is semantic-driven, therefore, the service wrapper designer retrieves the domain-specific types from the Catalogue 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 by the designed Resource Adapter will be compatible for pre-conditions of subsequent filter steps and or screens.

We may provide some analysis tool? is it implemented? otherwise I don't see it's worth to point this out, or maybe just in a "future work" section - Ismael

3.4   Generating a Resource Adapter

Once the wrapping of a service has been defined and tested in the Service Wrapper Tool, it generates an implementation of the desired Resource Adapter in XML, HTML, and JavaScript, ready to be deployed and executed inside a gadget. In addition, a description of the new building block is added to the FAST Catalogue. Thereby, the new Resource Adapter becomes available for the construction of new screens and screen-flows.

XML, HTML and Javascript? I don't get why XML and HTML is generated. The service wrapper should be just a javascript encapsulation of the web service, isn't it? HTML files should be for forms and XML for specific platform configuration files for deployment? - Ismael

3.5   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. However, 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. This paragraph doesn't talk about limitations, these are features! - Ismael

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 hard-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 these complex cases.

3.6   Other approaches

There is a number of related work that we have consider to use for service wrapping instead of building a Service Wrapper Tool ourself. However, due to the special needs of the Gadget Development Process it turned out that we were not able to use an existing approach directly but we were only able to grab ideas from existing approaches and come up with our own implementation. As discussed, service wrapping requires three steps. First we have to define the connection between the desired service wrapper and the other building blocks of FAST. This is achieved by the definition of pre- and post-conditions in our tool. Since this is very FAST specific, finding an existing tool for this purpose did not work. However, this is a simple step anyhow requiring only limited implementation efforts.

Second, we have to construct the request and to send it to the target service. There is already a number of REST clients for this purpose. There are even web based REST clients that may be used in a browser. We could easily have used such a standard REST client. However, we considered the user interfaces of these REST clients as too complicated for our end users. In addition, we wanted an easy and simple connection to the pre- and post-conditions of the service wrapper. This required the integration of a template expansion mechanism into the REST client which was most easy to achieve with a new interface build by ourself. Again the effort for this step was limited and justified by the smooth integration into the overall tool.

Third, we needed a data mediation step. This might have been achieved by e.g. XSLT tools. XSLT tools work on XML input and provide a template based generation of textual output. The problem with XSLT is a quite complex syntax which is not adequate for our end users.Well, no more than writing a SOAP message :) - Ismael It would also have been difficult to provide our end users with direct feedback, as required by FAST's storyboard philosophy. Similarly, interactive help e.g. by selection boxes for target types and target attribute name would not have been possible, easily. Instead of XSLT we might also have used model based transformation languages like ATL (Atlas Transformation Language) or TGGs (Triple Graph Grammars). While these approaches work on a higher level of abstraction compared to XSLT, we still consider them too complicated for our end users. In addition, it would have been rather difficult to achieve an integrated interactive support as we have achieved in our own implementation. However, our translation rules are heavily influenced by TGGs which was a tremendous help for coming up with a simple and easy to use yet powerful approach.

I couldn't really understand how a wrapper for a WSDL is constructed. The wrapper tool uses any library (Axis, etc) to parse the WSDL definition and extract the operations with their inputs and outputs? If so, it should be mention. Or should be the user who reads the WSDL document, checks the endpoint URL for the operation that wants to wrap, and then construct the wrapper as in the RESTful case, writting the SOAP message for the request body? - Ismael

4 Related Work

The rule based approach of the service wrapper tool has been inspired by triple graph grammars, cf. [**?**, Jahnke et al., 1997]. While general triple graph grammars allow to relate complex graph structures with each other, in our case the target of a rule is always a single object or attribute. This facilitates the whole mechanism, reasonably.

Write about some related work such as:

Wrapping WSDL-Described Web Services as Moby Services (using SAWSDL) `http://biomoby. open-bio.org/CVS_CONTENT/moby-live/Java/docs/sawsdlServlet.html`

is this section done? - Ismael

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.

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

[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)*.

[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).

Appendix A (Lists of Tables and Figures)

List of Tables

List of Figures