# AUTOMATIC ORCHESTRATION OF WEB SERVICES THROUGH SEMANTIC ANNOTATIONS

Philippe Larvet
*Alcatel CIT, Research and Innovation, Route de Nozay, 91461 Marcoussis, France*
*Philippe.Larvet@alcatel.fr*

Keywords: web service, business process, orchestration, service composition, semantic web service, semantic annotation.

Abstract: A new service can be developed as an orchestrated composition of existing web services. This paper describes an original process to automate the composition of semantic web services, by processing their "semantic tags". These tags can be extracted from the WSDL descriptions of the services and inserted into a light semantic description attached to the operations of the considered web services. A specific mechanism can examine these tags and determine automatically the possible "connectivity" of two given web services: the output of WS1, for example, semantically fits with the input of WS2. Then, the two web services are semantically connectable. This process can be used within the context of a service creation environment, in which the developer often wishes to assemble different services corresponding to an initial request. By using the semantic tags, a specific composition mechanism is able to connect automatically the chosen services and to assemble them to produce the final service that fits with the original request.

## 1 PROBLEMATIC OF WEB SERVICES DEVELOPMENT

Web services, as they are often stateless and contextless pieces of software, accessible from any point of Internet, are more and more suitable and convenient to build light and reusable applications. Globally, from the point of view of their internal complexity, web services (WS) can be divided in two families : *elementary* WS and *composite* WS.

The elementary ones provide a basic service, comparable to mathematical libraries, and contain a low level of data transformation, embedded in few algorithms ; for example, translation services are elementary WS.

On the contrary, the composite WS are able to provide a high level service and contain several levels of data accesses and transformations, given by the cooperation of several elementary services. For example, reservation services or secured-payment services are samples of composite WS.

If elementary Web Services can be built and relatively easily deployed with standard environments like Java with Apache/Axis or C# with .NET platforms, it could be interesting to have at one's disposal a powerful mechanism to *compose* WS as aggregations of existing services.

The main problem addressed by this desired mechanism is to express the aggregation of the legacy services, their interaction and the way they have to run in order to reach their objective and to provide the final service.

Several composition techniques exist today, and even if the industry is not yet agree on a common language, there are two languages that are considered as complementary:

- WSBPEL (Web Services Business Process Execution Language) or BPEL (BPEL, 2005), (Kavantzas, 2003): it describes the interactions between web services, including business logic and order of the interactions
- WS-CDL (Web Services Choreography Description Language) (Kavantzas, 2004): it describes the messages exchanged between web services, including order and constraints on these exchanges.

Like BPEL, we focuse in this paper on the description of the orchestration (Peltz, 2003). But BPEL is not the

only way to describe a business logic: Java or C# can also be used.

With SAMPAN, a multi-actor and agnostic Simple & Agile Method and Platform for service AggregatioN and deployment (Larvet, Bonnin, Ferres, Fontaine, 2005), we have proposed in 2005 an original way to solve this problem: an orchestration is derived from a constrained natural language description of the requested service.

All these solutions are based upon a formal (in BPEL or CDL) or pseudo-formal (in SAMPAN) description of the requested composite service. But, if we add a suitable description to the services to be composed, would they not become able to make themselves their composition? This is the challenge we propose to solve in this paper.

## 2 SEMANTIC WEB SERVICES COMPOSITION

A service contains several operations, and when we say *service composition*, in fact we mean *composition of the operations* that belong to their respective services.

The main idea to allow an automatic composition is to complete the description of the operations of the potentially composable services by adding to them some metadata that give useful and *semantic* information concerning the operations.

If these *semantic* informations are suitably chosen and set, the connectivity of the services becomes possible. For example, if the output of `WS1.operation_A()` semantically fits with the input of `WS2.operation_B()`, then WS1 can be composed with WS2, through the link "output of A" to "input of B", and we are authorized to write something like:

```
out_A=WS1.operation_A(A_parameters);
out_B = WS2.operation_B(out_A);
```

or, more directly:

```
out_B=WS2.operation_B(WS1.operation_
A(A_parameters));
```

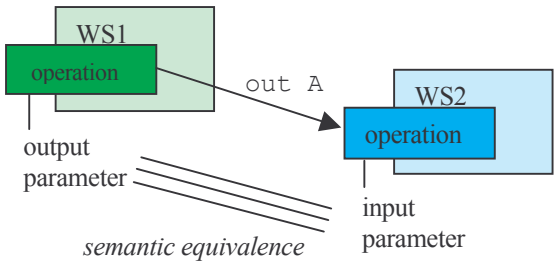


Figure 1 – Connectivity of two services through a semantic equivalence between inputs and outputs

In this schematic example (see Fig.1) we consider WS1 is connectable to WS2 through the operations `WS1.operation_A()` and `WS2.opration_B()` because the output of `WS1.operation_A()` is *semantically equivalent* to the input of `WS1.operation_B()`. Then, we can imagine an orchestration of WS1 and WS2 in which `WS2.operation_B()` is called just after `WS1.operation_A()` and consumes as an input the data provided by `WS1.operation_A()` as an output.

This connection is possible because of the *semantic equivalence* between the output of a given operation of the first service and another operation of the second one. What does that mean? It means that the two data have the same semantic "dimension", i.e. they semantically fit together - they are process-compatible. In other terms, they have not only the same data type, but the same "nature" of data.

For example, let us suppose `WS1.operation_A()` provides a text, and `WS2.operation_B()` is the Translate operation of a service Translator: it makes sense to translate a text, then the output of `WS1.operation_A()` has to fit with the input of Translator.Translate. But suppose now that `WS1.operation_A()` provides the stock value for a given company. This value and the text taken as input by Translator.Translate can have the same data type (String), they are not semantically equivalent, because it doesn't make sense to try to translate a stock value. Then, the semantic information attached to these two data must be different, and consequently the two operations are not connectable.

To make possible the *semantic connectivity* between two web services, we propose to attach to each parameter of each operation a semantic information that we call a *semantic tag*. This tag can be seen as a "tagged value", as defined in UML (Rumbaugh, Booch, Jacobson, 2004) and is intended to extend the semantics of the tagged parameter.

The semantic tags are set in a formal XML description of the web service, derived from its WSDL, and that we consider as a light "semantic WSDL", but without the complexity of SAWSDL (W3C, 2006).

For example, the Fig.2 below shows the "Semantic Light WS Description" (SLWD) for the service Translator, in which the semantic tags have been mentioned.

```
<service name="Translation">
<URL>http://xxx.xx.xx.xx/services
/TranslationService.asmx</URL>
  <operation name="Translate">
    <input name="src_lang"
typ="string" semtag="language" />
```

```
    <input name="dest_lang"
typ="string" semtag="language" />
    <input name="text_to_
translate" typ="string"
semtag="text" />
    <output name="translated_
text" typ="string" semtag="text"
/>
  </operation>
</service>
```

Figure 2 – Semantic Light WS Description (SLWD) of Translator service, with semantic tags

Nota: within the context of our Alcatel projects, we use some other semantic constructs in SLWD, for example <goal> in order to describe semantically the goal of an operation. But intentionally we don't show here these details, because they are not used to help the automatic services composition.

## 3 SETTING SEMANTIC TAGS IN WEB SERVICES DESCRIPTIONS

The web service description written in SLWD, as shown in Fig.2 above, can be generated from the WSDL. But how to set automatically the semantic tags? In the scope of our projects, we use a specific Semantic Module to do it.

This Semantic Module analyzes the names and types of the operations' parameters, as described in WSDL, and searches semantic correspondances in a specific ontology.

This ontology contains the links between the semantics of the current names and types of input and output data, as they are usually used by programmers, and the corresponding semantic tags.

For example, a data named "text" or "content" or "translated_page" or "description", with the type "string", will have the semantic tag "text" – because the data has the "dimension" of a text. A data named "date" or "current_date", with a type "Date" or "String" will have the semantic tag "date", etc.

This ontology can be expressed as a simple correspondence table, as shown in Fig. 3 below.

| **Data name** | **Type** | **Semantic Tag** |
|---|---|---|
| text, content, page, description, … | String | text |
| date, current_date, … | String \| Date | date |
| phone_number, mobile_phone, … | String | telephone_number |
| lang, language, dest_lang, srce_ lang, … | String | language |
| postal_code, zip_code, city_code, … | String | zip_code |
| … | | |

Figure 3 – Ontology for automatic setting of semantic tags in WS light descriptions

Such an ontology is easy to build and to improve little by little, by analyzing the contents of published WSDL that show the practice of programmers and then, by summarizing their "good usages".

## 4 AUTOMATIC SEMANTIC WEB SERVICES COMPOSITION

Let us take an example to describe the process that takes into account the semantic tags in order to build an automatic orchestration of web services.

We want to compose a new service, from a user's request: "I want a french version of the latest news from Reuters". This request could be expressed into a formal language or directly in natural language – this is outside the scope of the present paper.

The analysis of the request determines the needs of services able to cover the query and a specific Discovery mechanism has to search – and to find – the available services.

Within the context of our example, let us suppose that two main services have been discovered: a RSS service and a Translation service. The RSS service aims to gather informations from RSS feeds accessible on Internet, and contains two operations: GetRSSTitles allows to get the main titles of the feed for a given URL, and GetDescriptionOfTitle allows to get the text of the news that details this title. The Fig. 4 below shows the SLWD for this service.

```
<service name="RSS_Service">
<url>http://xxx.xx.xx.xx/services/
RSS_Service/RSS_Service.asmx</url>
    <operation
name="GetRSSTitles">
      <input name="adr_site"
typ="string" semtag="URL" />
      <output
name="list_of_titles"
typ="string[]" semtag="title" />
    </operation>
```

```
    <operation
name="GetDescriptionOfTitle">
      <input name="site_address"
typ="string" semtag="URL" />
      <input name="title"
typ="string" semtag="title" />
      <output name="description"
typ="string" semtag="text" />
   </operation>
</service>
```

Figure 4 – Semantic Light Description for the RSS Service

The Translation service is a classical one, that transforms a text (given as an input parameter) written in a given source language (input) into a translated text (output) written in a destination language (input). The SLWD of this service is shown in Fig.5 below:

```
<service name="Translation">
<url>http://172.25.75.xx/services/
TranslationService/Translation.asm
x</url>                          URL
  <operation name="Translate">
    <input name="src_lang"
type="string" semtag="language" />
    <input name="dest_lang"
type="string" semtag="language" />
    <input
name="text_to_translate"
type="string" semtag="text" />
    <output name="translated_text"
type="string" semtag="text" />
  </operation>
</service>
```

Figure 5 – Semantic Light Description for the Translation Service

Now, the problem is to compose automatically these two services – these three operations (see Fig.6) – in order to cover the original request: *provide a french version of the latest CNN news*.

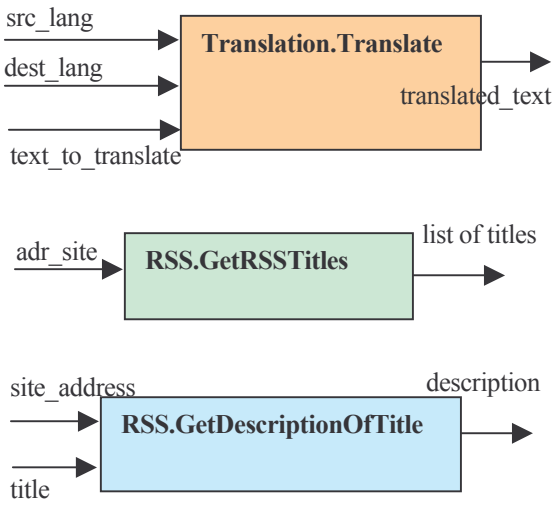


Figure 6 – How to compose automatically these 3 operations ?

For an automatic orchestration, the first key is to see the semantic tags as inputs and outputs of the operations. Then, some possible connectivities appear (see Fig.7), but not precise enough to make a full-consistent composition.

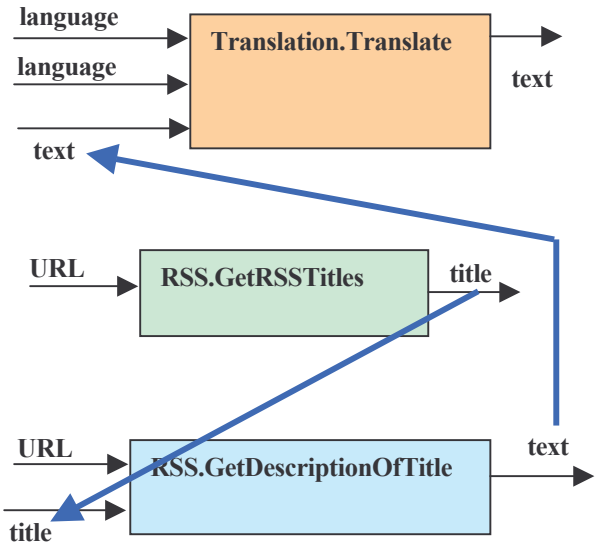


Figure 7 – Some possible connections (in blue) appear thanks to the semantic tags

The second key – the real bootstrap of the process – is to consider the *main output* of the targeted composed service, to search which operation(s) can provide its inputs, and to iterate the same process for this(ese) operation(s): search which other operation(s) can provide its(their) inputs. Then, progressively we go back from the main output to the input data necessary to produce it, and doing this, we assemble automatically the different operations by linking their outputs and inputs. In the same time, we can write

these links in a FILO stack (first in, last out) under the form of a pseudo-code expressing the calls of operations. At the end of this process, the content of the stack represents the orchestration of the new targeted service.

The main output of the targeted service is given by the expression of the original request. For our example, one wants a *translated version*: the main output is a translated text, i.e. the output of the operation `Translation.Translate`. We can write in the stack this main output, expressed as the "return" of the function represented by the targeted orchestration:

```
translated_text =
Translation.Translate(src_lang,
dest_lang, text_to_ translate);
return translated_text;
```

Then, we go back now to the inputs of this operation, whose the semantic tags are "language", "language" and "text". A data with a semantic tag "text" is provided by `RSS.GetDescriptionOfTitle`, then we can connect this operation to `Translation.Translate`: it means that we can add in the stack the call of operation `RSS.GetDescriptionOfTitle`, making the link with `Translation.Translate` through the name of the exchanged parameter:

```
text_to_translate =
RSS.GetDescriptionOfTitle(site_
address, title);
translated_text =
Translation.Translate(src_lang,
dest_lang, text_to_ translate);
return translated_text;
```

Now, we go back to the inputs of `RSS.GetDescriptionOfTitle`, whose the semantic tags are "URL" and "title". A data with a semantic tag "title" is provided by the operation `RSS.GetRSSTitles`, and then we can connect also these two operations, by pushing a new operation call in the stack:

```
title =
RSS.GetRSSTitles(adr_site);
text_to_translate =
RSS.GetDescriptionOfTitle(site_add
ress, title);
translated_text =
Translation.Translate(src_lang,
dest_lang, text_to_ translate);
return translated_text;
```

All the "discovered" services being used and connected together, the stack contains now the general texture of the orchestration. However, this texture must be refined before to be executed:

- the data types must be taken into account; for example, `RSS.GetRSSTitles` returns an array of String and not a single String;
- some parameters can be solved with some useful informations contained in the original request; for example, one wants a *french* translation, then the parameter "dest_lang" of the operation `Translation.Translate` can be set to "french";
- some other services can be used to solve other parameters; for example, the parameter "src_lang" can be set by using a utility service, a "Language Finder", to determine automatically the source language of a given text.

A specific module, whose the detailed description is outside the scope of this paper, makes these refinements in order to complete the pseudo-code:

```
String[] Orchestration(String
site_address) {
  String[] result;
  titles =
  RSS.GetRSSTitles(site_address);
  foreach title in titles {
  text_to_translate =
  RSS.GetDescriptionOfTitle(site_
  address, title);
  source_lang =
  LanguageFinder.GetLanguage(text
  _to_translate);
  translated_text =
  Translation.Translate(source_la
  ng, "french",
  text_to_translate);
  add to result title +
  translated_text;
  }
  return result;
}
```

This pseudo-code can be finally transformed into an executable BPEL file, for example, and transferred to a BPEL engine, or it can be translated into C# or Java and deployed as a new web service in Microsoft IIS or Apache/Axis environments.

## 5 PERSPECTIVES AND WORK IN PROGRESS

Today, the building of the correspondence table used to add the semantic tags to the light web services semantic descriptions (see Fig.3) is still partially manual. This table comes from the analysis of the WSDL content of our published web services.

We are currently working on a semantic module able to expand the names of the operation parameters found in WSDL and to search these expansions in external ontologies, in order to discover their semantics. For example, "lang" could be expanded into "language", "src" could become "source", etc. and this clarification allows a better search of the meaning of the term in appropriate ontologies.

Another effort is made on the composition process itself. In some cases where more than three operations have to be composed together, some unexpected loops or dead ends can occur; in other cases, a mediation between data is necessary, for example to connect an operation using a date expressed by three parameters "day", "month", "year" with another operation where "date" is only one parameter "dd/mm/yyyy". These kinds of cases demand a more effective composition module, on which we are currently working today.

## 6 CONCLUSION

We have tried to show in this paper a new strategy to compose automatically web services by using simple semantic annotations.

This strategy has several advantages:

- it is simple to implement: an adapted pre-processing can easily build a light description of a web service from its WSDL, and a post-processing, using a simple correspondence table, can complete this description by setting semantic tags;
- it allows the processing of formal requests (or even natural language requests), where the user expresses the final service he wants; the processing of a formal request allows to determine (to discover) the pertinent services to be composed;
- it allows the automatic composition of a set of services that are given in any order; the logical order of the composition – the order of the operations calls – is determined with the help of the semantic tags, and with the logic of the original request;
- it can give the possibility to compose on-the-fly some on-demand services, then it allows to respond dynamically to the user's requests.

Inserted in a more general process – request analysis, service discovery, automatic composition, final service deployment and delivery – this strategy helps to build a consistent orchestration, ready to be generated into BPEL, C# or Java to become the new service wished by the user.

**References**


BPEL, 2005. BPEL Editorial Team, *BPEL Learning Guide,* February 2005, http://searchwebservices.techtarget.com/originalContent/0,289142,sid26_gci880731,00.html

BPMI, 2002. BPMI, Business Process Management Initiative, *BPML, Business Process Modeling Language Specifications,* BPMI.org, 2002, http://www.bpmi.org/specifications.htm

Kavantzas Nickolaos & al., November 2004. *Process-centric realization of SOA : BPEL moves into the limelight,* Web Services Journal, http://www.findarticles.com/p/articles/mi_m0MLV/is_11_4/ai_n7071401

Kavantzas Nickolaos, Dec. 2004. *WS-CDL, Web Service Choreography Description Language,* http://www.ebpml.org/ws_-_cdl.htm and http://www.w3.org/TR/ws-cdl-10/

Larvet Philippe, Bonnin Bruno, Ferres Lamia, Fontaine Patrick, 2005. *A Multi-Actor Agnostic Platform for Web Services Agile Development and Deployment,* ICSSEA 2005, Vol.2, Sessions 9-16

Rumbaugh J., Booch G., Jacobson I, June 2004. *UML Reference Manual, Second Edition,* Addison-Wesley.

W3C, January 2007. SAWSDL, *Semantic Annotations for Web Service Description Language,* http://www.w3.org/TR/sawsdl/

Dubray Jean-Jacques, June 2004. *BPML for Web services*, in http://www.ebpml.org/bpel4ws.htm

Nanda Mangala & al., Nov. 2004. *Decentralized Orchestration of Composite Web Services,* IBM Research Computer Science, Innovation Matters, http://www.research.ibm.com/compsci/project_spotlight/distributed/

Peltz Chris, Jan. 2003. *Web services orchestration, a review of emerging technologies, tools, and standards,* Hewlett-Packard Co, http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf

Smith Howard, July 2003. *BPM and MDA, Competitors, Alternatives or Complementary,* Business Process Trends, White Paper, http://www.bptrends.com/publicationfiles/07-03%20WP%20BPM%20and%20MDA%20Reply%20-%20Smith.pdf