RAWS Architecture: Reflective and Adaptable Web Service Model

Javier Parra-Fuente[1], Salvador Sánchez-Alonso[2], Oscar Sanjuán-Martínez[1], Luis Joyanes-Aguilar[1]

[1]*Pontifical University of Salamanca, Madrid campus, SPAIN*
[2] *University of Alcala, Madrid, SPAIN*

ABSTRACT

Web services are static components, which implies that before a change in their structure or behavior can be made, the source code — or a decoder of compiled code — is needed. The full process consists of three steps: editing and modifying the source code, compiling it again, and redeploying it in the server. Reflection, a powerful tool for the adaptation of applications at runtime, may help in creating more flexible and dynamic Web services. In this paper, we introduce RAWS (Reflective and Adaptable Web Service) Architecture, a Web service design model based on a reflective architecture multilevel. RAWS allows both the dynamic modification of the definition and implementation structure of the Web service, and the dynamic modification of the Web service behavior in order to change the existing code or to add new functionalities. All these dynamic modifications are performed directly on the code during execution, with no need to have the Web service source code. We also introduce an automatic generator of the reflective infrastructure that is needed for the implementation of the RAWS architecture. This infrastructure will make possible that any Web service can automatically behave like a Reflective and Adaptable Web Service.

Keywords: *adaptability; architecture; behavioral reflection; customization; introspection; maintainability; structural reflection; Web service*

INTRODUCTION

Web services are programmable components of applications that use SOAP (Gudgin et al., 2003) as an access protocol, regardless their client and component technology (a drawback in DCOM) and regardless the language in which both communication ends are written (a drawback in RMI). SOAP generally uses the HTTP transport protocol, over the port 80 for request/response, thus crossing corporate firewalls (a drawback in CORBA or DCOM) and facilitating the interoperability of applications that work with different technologies.

Currently, the modification of a Web service implies the availability, edition, recompilation and redeployment of the source code. Depending on the application server, the deployment task can be a simple or a complicated one. If the application server supports the dynamic load of applications, then the deployment will be simple task; but if that is not the case, it will imply to stop the execution of the Web service, replacing the old version with the new one, and deploying the new version.

Reflection is a property of computational systems that allows them to reason and act by themselves and to modify their behavior (Maes, 1987). Although this concept has been successfully applied to other fields such as distributed systems (Ledoux, 1999; McAffer, 1995), concurrent programming (Masuhara, Matsouka, & Yonezawa, 1993), aspect-oriented programming (Pawlak, Duchien, & Florin, 1999; Tanter et al., 2003), and etcetera, its application to Web service design has not been addressed yet.

Reflection can be applied to Web services in order to enhance their adaptability and flexibility. We propose in this paper a Web service reflective architecture, RAWS, which allows one to dynamically modify a Web service during its execution.

In this paper, we introduce the basic concepts of reflection that will be applied to Web services (the introspective characteristics and the analysis of the structural and behavioral reflection of the Web service), the architecture model of a reflective and adaptable Web service, and the automatic generation mechanism to obtain the reflective infrastructure needed for a Web service to be dynamically adaptable.

REFLECTIVE WEB SERVICES

In general terms, reflection in a system can be classified into three groups, depending on the information that the system can reflect:

- *Introspection* (Foote, 1992): the system is able to observe and reason on the system elements, but it is unable to modify them.
- *Structural Reflection* (Foote, 1989): the system can enquire and modify its structure at runtime.
- *Behavioral Reflection* (Ferber, 1989): the system can manipulate and modify its behavior.

The RAWS architecture will prove how all the aforementioned kinds of reflection can be successfully applied to Web service design. Reflection is usually represented, as in *Figure 1*, by a two-level architecture: a base level that contains the modules that solve the problem, and a meta-level containing the representation of the base level. An application is represented in the base level and can be manipulated by the meta-level. Both levels are joined together by a causal connection (Smith, 1984), so that the changes brought about in the base level are reflected in the metalevel.

On the other hand, in order to manipulate the information of the meta-level, the computational behavior of the base level object is transformed into data. This process is called reification (Smith, 1982). The reificated information makes up the metainformation, thus allowing the reflective behavior.

The association of both the base and meta-levels and the implementation of reflection can take place in two ways: (a) *explicitly*, if the base level object activates the reflection, or (b) *implicitly*, if the system activates the meta-object.

As already stated, the abovementioned architecture can be applied to Web service design in order to enhance flexibility and adaptability. In the case of Web services, the base level will contain the Web service itself, while the metalevel will contain the meta-representation of the Web service.

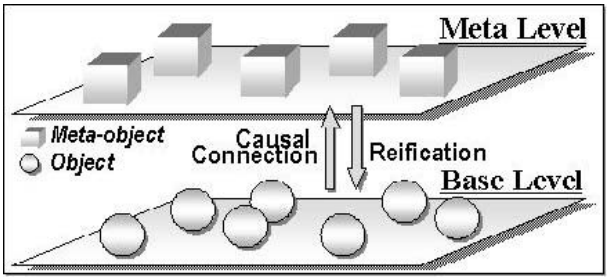

Figure 1. Reflective system architecture

INTROSPECTIVE NATURE OF WEB SERVICES

Although introspection allows one to consult the structure of a given system, the way it communicates, and etcetera, yet it does not allow its modification. Web services basically use three XML standards: SOAP (Simple Object Access Protocol), WSDL (Web Service Description Language) and UDDI (Universal Description, Discovery and Integration). XML documents have a hierarchical structure of elements, attributes and entities, using Document Type Definition (DOM) (Stenback, Le Hégaret, & Le Hors, 2003) or schemas (Fallside, 2001) to describe the grammar of each specific document. This hierarchical structure facilitates the introspection in

XML documents.

Introspective Description of a Web Service

WSDL documents (Christensen et al., 2001) describe the way that the business methods and the physical location of a Web service can be accessed. These documents contain five introspective characteristics of Web services: types, message, port type, binding and service.

We will be able to gather information about the structure of the Web service and the way it works by using introspection in its WSDL document. As an illustration, let us think of a Temperature Web service that has been located by a client. The client needs to know both how to use it and the structure of its business methods before it can be used. In this way, an introspective Temperature Web service would allow the client to ask for directions on how to use it, and then be able to fully use it.

Introspective Communication of a Web Service

SOAP documents describe the format of the messages to be sent among the various Web service participants (provider, client and broker). These messages can be classified into the following types: request, response, error and data transfer.

In some given situations, either the client or the Web service itself needs to know about the contents of a certain SOAP message. SOAP messages are based in XML, which makes it easier to design interfaces that handle these message contents. In this way, several Java-based APIs like JAXM and JAX-RPC have been proposed.

Introspective Location and Publication of a Web Service

UDDI (Bellwood, Clément, & Von Riegen, 2003) is a service for locating Web services for clients as well as an advertising mechanism for Web service brokers. The advertisement of a Web service is made by means of information models that are defined using XML. There are four types of introspective information: business entity, business service, binding template and tModel.

Among all the aforementioned Web service standards, UDDI is the one that more clearly addresses introspection, since it is aimed at the publishing and discovering of Web services. The introspective features of UDDI registries make it possible to know about the broker company, the binding points, as well as the way of accessing a Web service, and etcetera.

STRUCTURAL REFLECTION OF A WEB SERVICE

The structural reflection of a Web service allows for the modification of its structure at runtime. Although structural reflection has been described in general terms and used in several projects (Gudgin et al., 2003; Kiczales, Des Rivières, & Bobrow, 1992; Maes, 1987), Web service structural reflection has not explicitly

been addressed yet. A reflective Web service presents two types of reflection: structural reflection of the Web service definition and structural reflection of the Web service implementation.

Structural Reflection of a Web Service Definition

Every Web service is self-describable due to its WSDL document, which describes its structure, business methods and location, among other features. In order to modify the structure of this description, and consequently the structure of the Web service, a meta-level that contains a representation of the Web service might be designed. We will refer to it as the Web Service Meta Description Level (WSMDL). This meta-level contains the representation of the Web service declaration (Meta-WSDL) and it does not depend on the programming language in which the Web service has been implemented.

In order to be able to work with Meta-WSDL we have developed a JRWSDL (Java Reflective WSDL) API, made up of 24 classes and interfaces that shall represent the description of the Web service. Each of these classes represents the meta-model corresponding part of the actual structure of the Web service. *Figure 2* shows a view of the most significant classes in this API.

In this model, as in a WSDL document, a Web service definition has a type, and it is composed of one or more messages, bindings, port types and services. Port types and services are in turn made up of operations and ports respectively. This API allows one to dynamically modify the Web service description characteristics, regardless its implementation. In this model, each API class represents a tag in the WSDL document, while the sub-tags are represented by attributes of such class.

In our previous Temperature example, this could be attained by representing each WSDL tag by a JR-WSDL class like the one shown in Example 1.

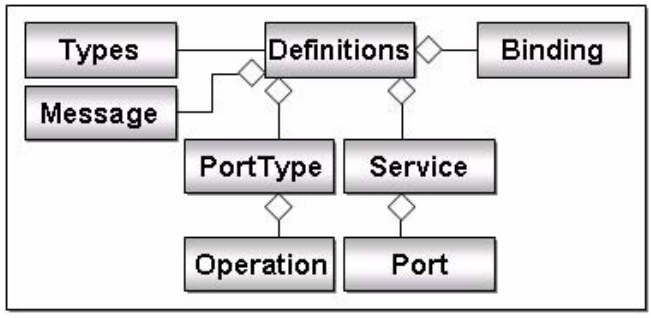

Figure 2. Partial class diagram of JR-WSDL API

Structural Reflection of a Web Service Implementation

The structural reflection of a Web service implementation is what allows one to modify the code structure of the Web service at runtime (names of business methods, types of attributes, access permissions, etc.). Following the model already described for a structural reflection of the Web service definition, we have developed a meta-level called Web Service Meta Implementation Level (WSMIL) for the

design of structural reflection. This metalevel contains Meta-Web services that are the representation of the Web services.

Meta-Web services are automatically generated from the Web service, and they contain the reflective operations that can be run on the Web service code. In order for the Meta-Web service to be able to modify the Web service structure, it needs to access its Meta-Code, which holds the representation of the Web service code being executed. This Meta-Code is language independent, and it is aimed at working on a virtual machine with a dynamic class loader. Although some specific implementations for other languages and platforms such as .NET or Smalltalk can be developed, we intentionally focus our work on Java.

In a java-based model, the Meta-Code of a Web service can be automatically obtained from the Web service itself by using the Java Reflection API and a structural and behavioral reflective API, that is, BCEL (Dahm, 1999). In such a reflective system like this one, reification of the base level on the meta-level is needed to generate Meta-Code from the original Web service. Authorized clients will invoke the Meta-Web service, which will in turn execute the reflective operation by using the Meta-Code of the Web service. On the other hand, in order for the Meta-Code to manifest the changes in the original Web service, a causal connection must be established in order to transfer the new code to be loaded from the meta-level to the base level. In our system, the new code is dynamically loaded by means of a user-defined class loader of the Java virtual machine. This implies that a custom user-defined class loader has to be created, and that the former, which contains the old version, has to be destroyed.

Example 1.

```
 <definitions>
    .....
    <message name="getTemperatureRequest">

<part name="city" type="xsd:string"/>
    </message>
    <message name="getTemperatureResponse">

<part name="getTemperatureReturn" type="xsd:string" />
    </message>
    <portType name="Temperature">

<operation name="getTemperature"...>
   <input message="getTemperatureRequest".../>
   <output message="getTemperatureResponse".../>


</operation>
    </portType>
    .....

 </definitions>
```

Uses the classes:

```
class PortType{
   private String name;
   private Vector operations;
  ...
```

```
    public String getName()
    public void setName(String)
    public Vector getOperations()
    public void setOperations(Vector)
    public void addOperation(Operation)
    public void removeOperation(int)

}

class Operation{
    private Vector messages;
   ...
    public void addInputMessage(Message)
    public void addOutputMessage(Message)

}
```

And corresponds to the code:

```
Message m1=new Message("getTemperatureRequest");
Message m2=new Message("getTemperatureResponse");
Operation op=new Operation("getTemperature");
op.addInputMessage(m1);
op.addOutputMessage(m2);
PortType pt=new PortType("Temperature");
pt.addOperation(op);
```

BEHAVIORAL REFLECTION OF A WEB SERVICE

Behavioral reflection is the Web service ability to dynamically modify its behavior. This can be observed from two different points of view: the modification of the code to be executed by the Web service, and the extension of the code to be executed, keeping part of the original code. Modifying a Web service can be done by using two different deployment approaches:

- *Static*: this approach involves editing the existing code, recompiling it, and then redeploying it in the application server. This model assumes that the source code is available, which constitutes one of its major disadvantages. Another important shortcoming is the difficulty of automating the modification process, because most code editions need human intervention.
- *Dynamic*: this model entails the transformation of the original Web service into a pair Web service – Meta-Web service, where the Web service can access its Meta-Web service in order to modify either its structure (structural reflection) or its executable code (behavioral reflection) or both. The dynamic deployment has, among other advantages, to automate the modification process. At the same time, source code availability is not needed since a meta-representation of the code in execution is always loaded in memory.

In the Temperature Web service example, the original service is transformed into two abstraction layers: the Temperature base Web service and the Temperature Meta-Web service as shown in *Figure 3*. Let us suppose that our original Temperature Web service is a local service aimed at providing information on temperatures in Spanish cities. Consequently, the response message returned by a query on a city temperature is written in Spanish. If we want this service to provide

information in English, we accordingly need to modify the method getTemperature() as shown in Example 2. This original Web service will be transformed into a base Web service (Example 3). This base Web service uses several meta-classes (Example 4). Finally, the client can modify the Web service source code by sending the message in Example 5 in a SOAP request.

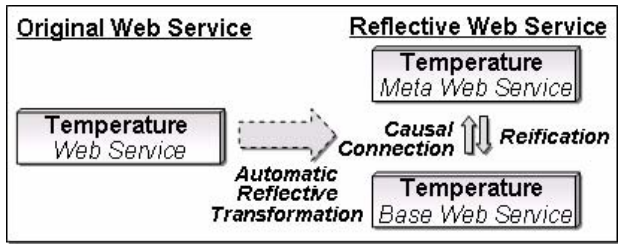

Figure 3. Transforming a Web service into a reflective Web service

Example 2.

```
public class Temperature{
        public String getTemperature(String city)

throws java.rmi.RemoteException{
           // SELECT into DataBase
           return "La temperatura en "+city+" es "+temp;

      }
}
```

Example 3.

```
public class Temperature{
            private Repository repository;
            private Vector methods;
            ...
            public String getTemperature(String city)

throws java.rmi.RemoteException{
           Method
      m=methods.getMethod("getTemperature");
           return m.execute();


        }

      public boolean modifyMethod(String name, byte
      code[]){
                Method
      m=methods.getMethod("getTemperature");
                m.setCode(code);
                ClassFile cf=reify();

      repository.destroyWSClassLoader("Temperature"
      );
                repository.createWSClassLoader(cf);
                return true;

      }
}
```

Example 4.

```
public class Method{
      private String name;
      private String returnType;
      private Vector parameters;
      private Code code;
      ...
      public void setCode(Code){...}
      public Code getCode(){...}
      public void excecute(){...}


}
```

Example 5.

```
<modifyMethod
     soapenv:encodingStyle="http://schemas.x
     mlsoap.org/
     soap/encoding/" xmlns:ns1="operation">
      <name xsi:type="xsd:string">
         getTemperature
      </name>
  <code xsi:type="soapenc:Array"
 soapenc:arrayType="xsd:byte[200]"

 xmlns:soapenc="http://schemas.xml
     soap.org/soap/encoding/">
     New code to update the original one
     <item>99</item>
     .....
     <item>104</item>

      </code>
</modifyMethod>
```

DYNAMIC GENERATION PROCESS

To have an adaptive behavior, a Web service needs a reflective infrastructure. In order to simplify the transformation from a “regular” Web service to an adaptable Web service, this task can be automated. The design of the automatic transformation shown here is platform-independent, which has two main advantages:

- Interoperability between Meta-Web services.
- Web service automatic translation from a computer language to another.

Figure 4 shows the different stages in the reflective infrastructure dynamic generation process.

Parsing Web Services

A source code parser has to be designed to identify the structure and behavior of the Web service. This parser depends on the implementation language of the service, because the syntax of each language is, of course, different. Thus, we would have a parser for Java, another one for C#, Smalltalk, and etcetera. The parser includes an API that reflects the Web service structure . Some representative classes of this API are those in *Figure 5*.

For example, the Temperature Web service described before has a business method called getTemperature() that returns the temperature of the parameter city. The structural parser analyzes each different word in the source code, identifying the different syntactic elements and consequently generating an object, attribute, and etcetera of the parser API for each one. When this analysis is finished, the structural parser generates a Class object containing the class name in the name attribute and the getTemperature method features in the method attribute. This getTemperature object has a public value in the access attribute, a string value in the return attribute, a Parameter object with the characteristics of the parameter city, and the method code is stored in a buffer that is an instance of the Code class.

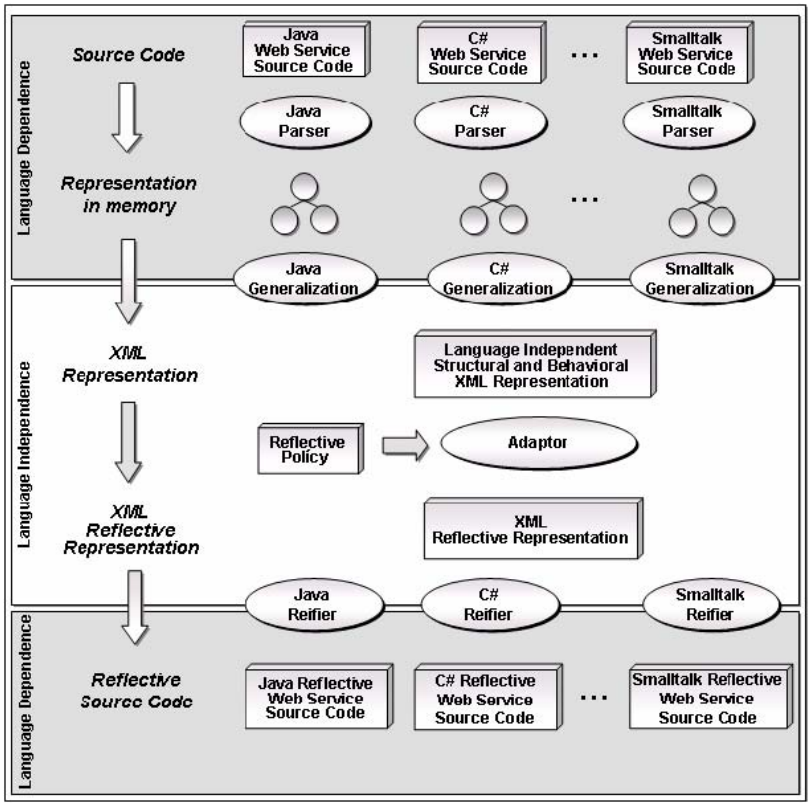

Figure 4. Dynamic generation of reflective infrastructure

Language-Independent Web Service Structure and Behavior Representation

When the source code has been completely analyzed, the next process is what we

call “generalization,” a process consisting in the abstraction of the Web service source code of the language in which it is implemented in order to obtain language-independent information. We have developed OOWSML (Object Oriented Web Service Markup Language), a language aimed at representing the structure and behavior of the Web service that deliberately avoids the syntactic details of the particular programming language used. *Table 1* shows the main elements and attributes that conform the grammar of this language.

Using the Temperature Web service analyzed above, its structure and behavior can be transformed so as to be language-independent. As a result of this process, the document written in OOWSML would be generated (Example 6).

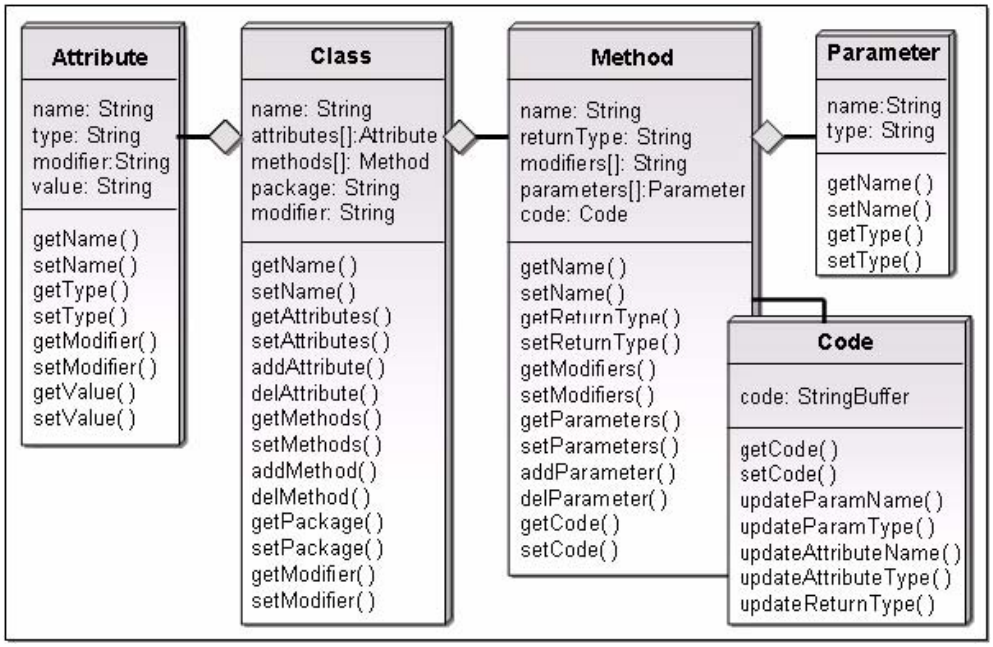

Figure 5. Parser API for the recognition of the structure

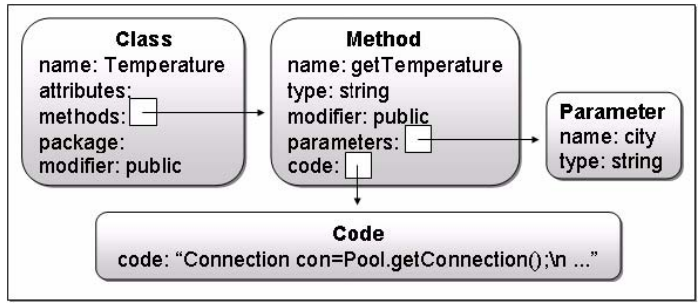

Figure 6. Objects generated by the Parser

Adding Reflective Features to a Web Service

The reflection level to implant in a Web service can be personalized by making use of the Web Service Adaptive Policy document. Using the objects returned by the Web service code analyzer as the starting point, the desired reflective infrastructure is subsequently introduced depending on the adaptive policy document. *Table 2* shows the reflective capacities that can be added to a Web service.

This table relates the properties of the Web service and the actions that can be done regarding those properties. When an "X" appears in the table, it must be read as: "it is allowed to perform the action in this column for the property in this row." In particular:

Elements	Attributes
application	name
package	name
import	name
class	name, visibility, abstract, static, final, extends
method	name, visibility, abstract, static, final, returnType, constructor
argument	name, type
attribute	name, type, value, visibility, static, final
block	--
simpleSelective	condition
else	--
multipleSelective	expression
case	value
loop	type, initial, condition, increment
var	name, type, value, static, final
assignment	var, value
methodCall	object, method
parameter	name
operation	op1, operator, op2
new	class
return	expression
break	--
continue	--

Table 1. Main elements and attributes

Equation 6.

```
<application
     name=" Temperature " >
        <class
     name=" Temperature "
     visibility=" public " >
      <method
     name=" getTemperature "
     visibility=" public"

                 returnType=" String "
   constructor=" false " >
   <argument name=" city "  type=" string" \>
   <block>

 ...  // SELECT
 into DataBase
   <return
 expression=" The
 temperature in
 " +city+ "  is
 " +temp />
 </block>
  </method>
    </class>
  </application>
```

Action / Property	get	set	add	delete
Name	X	X		
Package	X	X		
Access	X	X		
Attributes	X	X	X	X
Methods	X	X	X	X

Table 2. Reflective capacities

get: it is allowed to consult the value of the property
set: it is allowed to modify the value of the property
add: it is allowed to add to a new element to the property
delete: it is allowed to eliminate an element of the property

A class has a name, it can be included into a package, and it has an access modifier, attributes and methods. All of them can be consulted or modified by the service creator. In addition, it is possible to be granted to add or delete class attributes and methods. The attribute properties can be managed at a global level, by using the attributes property if all the attributes have the same properties, or at individual level, if each attribute has different properties. In this case, each particular attribute will have the reflective characteristics shown in Table 3.

It can be consulted or modified, for each attribute, its name, type, access modifier and/or initial value. In the same manner as with the attributes, methods can be dealt with at global level by using the methods property, or at individual level, if it is allowed that methods have their own reflective properties (see Table 4).

The name, return type, parameters, access modifiers and code of the methods can be consulted or modified, and in the same way, its parameters or access modifiers can be added or eliminated.

Dynamically changing these properties sometimes will imply remaking the WSDL document of the Web service to ensure the integrity between the document and the Web service. When the reflective methods are added to the Web service, all the necessary functionalities to regenerate its WSDL document are automatically added as well, which will allow one to regenerate it at any time in the future.

Regenerating the Reflective Source Code

Reflective source code regeneration is made by using the OOWSML document, which represents the Web service. This regeneration is carried out by a reifier, which depends on the desired programming language of the final code. We should remark that the reflective Web service language could be different from the language of the original source code, thus transforming Web services automatically from one language to another. The reifier uses XSLT to generate the new reflective source code.

Action / Property	get	set	add	delete
Name	X	X		
Type	X	X		
Access	X	X		
Value	X	X		

Table 3

Action / Property	get	set	add	delete
Name	X	X		
Return	X	X		
Parameters	X	X	X	X
Access	X	X	X	X
Code	X	X		

Table 4

RAWS ARCHITECTURE

In this work, we propose the RAWS architecture, a multilevel reflective architecture integrated by different levels that communicate by means of causal connection and reification. These levels are:

- *Meta-Web Service Level*: it contains a meta-representation of the Web service. This representation, written in OOWSML, is programming language-independent.
- *Meta-Representation Level*: it contains the meta-representation of the Web service and includes both the necessary reflective methods to communicate the Web service with the Meta-Web service, and a reflective interface to remotely access/modify the Web service.

- *Web Service Level*: it contains the Web service with the code modifications to be able to communicate with its meta-representation.

Then, if a Web service can behave like a reflective and adaptable Web service, it will be necessary to transform it by generating its reflective infrastructure, as it was explained in the previous section.

RELATED WORK

RAWS Architecture is based on both a base level and a meta-level architecture, which has its origin in the systems based on Meta Object Protocols (MOPs) (Maes, 1987). There are various works related to MOPs and the dynamic modification of their semantics, such as Clossete (Kiczales, Des Rivières, & Bobrow, 1992), Cognac (Murata et al., 1994), Iguana (Gowing & Cahill, 1996), MetaXa (Golm & Kleinöder, 1998), Guanará (Oliva, García, & Buzato, 1998) or nitrO (Ortín & Cueva, 2001).

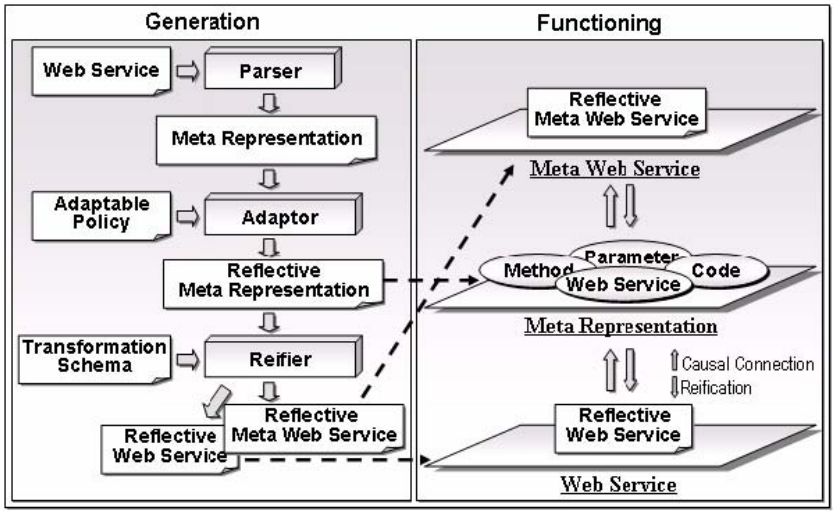

Figure 7. RAWS architecture

Web services technology is independent of the programming paradigm that is used in the implementation of the service. The RAWS system is based on the Object Oriented Paradigm (OOP), but it is designed to work on other paradigms based on the study of diverse reflective systems applied to diverse languages and programming paradigms: procedural paradigm -3-LISP (Des Rivières & Smith, 1984), BROWN (Friedman & Wand, 1984), functional paradigm -TEIRESIAS (Davis & Lenat, 1982), SOAR (Laird, Newell, & Rosenbloom, 1987), logical paradigm -FOL (Weyhrauch, 1980), META-PROLOG (Bowen, 1986), or OOP -3-KRS (Maes, 1987), and Smalltalk (Rivard, 1996).

FUTURE WORK

The RAWS Research Group, specifically created to develop the described model, is currently working on a .NET extension of the proposed architecture. Another important research topic is the design of an intermediate level between the Web

service and the Meta-Code. This intermediate layer will provide the meta-level with paradigm-independence. Our interest in automating the Web service modification process is also an outstanding issue that is currently being addressed. The design of a communication prototype for the clients to be able to remote and dynamically modify the web service is one our priorities.

CONCLUSION

In this paper, the RAWS architecture has been presented. It is aimed at the dynamic design of reflective and adaptable Web services, and establishes a three level model. The Web service level contains the conventional Web service and it is self-describable. The meta-representation level contains the meta-representation of the Web service, acting as an interface of reflective communication between the Web service and the Meta-Web service. The Meta-Web service level contains a platform-independent meta-representation: what makes the interoperability between different Meta-Web services possible.

The dynamic generation process of a Meta-Web service has been introduced in order to show how any Web service can behave like a Reflective and Adaptable Web Service by modifying the original code. This process includes a parser to analyze the source code and to obtain the structure and behavior of the Web service, an adaptor to add the reflective characteristics specified by the Web service author in the Adaptable Policy Document, and finally a reifier to generate both the reflective Web service and the reflective Meta-Web service.

The main contribution of this work is a model that allows designing flexible and adaptable Web services based on reflection. This model allows one to dynamically modify the structure of a Web service definition and implementation, as well as the Web service behavior.

REFERENCES

Bellwood, T., Clément, L., & Von Riegen, C. (2003). *UDDI Version 3.0.1*. UDDI Spec Technical Committee Specification, OASIS.

Bowen, K. (1986). Meta-level programming and knowledge representation. *New Generation Computing, 3* (4), 359-386.

Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001). *Web Services Description Language (WSDL) Version 1.1*. W3C Note 15, World Wide Web Consortium.

Dahm, M. (1999). Byte code engineering. In *Proceedings of Java Informations Tage (JIT'99)* (pp. 267-277). Berlin: Springer-Verlag.

Davis, R., & Lenat, R. (1982). Knowledge-based systems in artificial intelligence. New York: McGraw-Hill.

Des Rivières, J., & Smith, B.C. (1984). The implementation of procedurally reflective languages. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (pp. 331347). New York: ACM Press.

Fallside, D.C. (2001). *XML Schema*. W3C Recommendation, World Wide Web Consortium.

Ferber, J. (1989). Computational reflection in class based object-oriented languages. In *Proceedings of the 4th International Conference on Object Oriented*

Programming Systems, Languages and Applications (OOPSLA'89) (pp. 317-326). New York: ACM Press.

Foote, B., & Johnson, R.E. (1989). Reflective facilities in Smalltalk-80. In *Proceedings of the 4th International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'89)* (pp. 327-335). New York: ACM Press.

Foote, B. (1992). Objects, reflection and open languages. In *Proceedings of the Workshop on Object-Oriented Reflection and Metalevel Architectures (ECOOP'92)*.

Friedman, D., & Wand, M. (1984). Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (pp. 384-355). New York: ACM Press.

Golm, M., & Kleinöder, J. (1998). MetaXa and the Future of Reflection. In *Proceedings of the Workshop on Reflective Programming (OOPSLA'98)* (pp. 1-5). New York: ACM Press.

Gowing, B., & Cahill, V. (1996). Meta-object protocols for C++: The Iguana approach. In *Proceedings of the Reflection'96 Conference* (pp. 137-152).

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., & Frystyk, H. (2003). *Simple Object Access Protocol Version 1.2*. W3C Recommendation, World Wide Web Consortium.

Kiczales, G., Des Rivières, J., & Bobrow, D. (1992). The Art of the Meta-Object Protocol. *ACM SIGPLAN Notices*,*27* (2), 9.

Laird, J., Newell, A., & Rosenbloom, P.S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33 (1), 1-64.

Ledoux, T. (1999). OpenCorba: A reflective open broker. In *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection'99)* (pp. 197-214). Lecture Notes in Computer Science. (Vol. 1616). Berlin: Springer.

Maes, P. (1987). Concepts and experiments in computational reflection. Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'87). *ACM SIGPLAN Notices*, *22* (12), 147-155.

Masuhara, H., Matsouka, S., & Yonezawa,
A. (1993). Design an OO reflective language for massive-parallel processors. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'93)*. New York: ACM Press.

McAffer, J. (1995). Meta-level programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)* (pp. 190-214). Lecture Notes in Computer Science, (Vol. 952). Berlin: Springer.

Murata, K., Horspool, R.N., Yokote, Y., Manning, E.G, & Tokoro, M. (1994). Cognac: A reflective object-oriented programming system using dynamic compilation techniques. In *Proceedings of the Conference of Japan Society of Software Science and Technology (JSSS'94)* (pp. 109-112).

Oliva, A., García, I.C, & Buzato, L.E. (1998). The reflective architecture of Guanará. Technical Report IC-98-14. Sao Paulo, Brazil: Institute of Computing, State University of Campinas.

Ortín, F., & Cueva, J.M. (2001). Building a completely adaptable reflective system.

In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'01)*.

Pawlak, R., Duchien, L., & Florin, G. (1999). An automatic aspect weaver with a reflective programming language. In *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection'99)*. Lecture Notes in Computer Science (Vol. 1616). Berlin: Springer.

Rivard, F. (1996). Smalltalk: A reflective language. In *Proceedings of the Reflection'96 Conference* (pp. 21-38).

Smith, B.C. (1982). Reflection and semantics in a procedural language. Technical Report MIT-TR-272. Cambridge, MA: MIT Press.

Smith, B.C. (1984). Reflection and semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (pp. 23-35). New York: ACM Press.

Stenback, J., Le Hégaret, P., & Le Hors, A. (2003). *Document Object Model (DOM) Specification*, Version 1.0. W3C Recommendation, World Wide Web Consortium.

Tanter, E., Noyé, J., Caromel, D., & Cointe, P. (2003). Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems and languages* (OOPSLA'03) (pp. 2746). New York: ACM Press.

Weyhrauch, R. (1980). Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence, 13* (1,2).