HAL
archives-ouvertes.fr

From Object-Oriented Programming to Service-Oriented Computing: How to Improve Interoperability by Preserving Subtyping

Diana Allam, Hervé Grall, Jean-Claude Royer

▶ **To cite this version:**

Diana Allam, Hervé Grall, Jean-Claude Royer. From Object-Oriented Programming to Service-Oriented Computing: How to Improve Interoperability by Preserving Subtyping. Karl-Heinz Krempels and Alexander Stocker. WEBIST 2013 - 9th International Conference on Web Information Systems and Technologies, May 2013, Aachen, Germany. SciTePress Digital Library, pp.169-173, 2013, WEBIST 2013 - Proceedings of the 9th International Conference on Web Information Systems and Technologies, Aachen, Germany, 8-10 May, 2013. <hal-00800153>

HAL Id: hal-00800153
https://hal.inria.fr/hal-00800153

Submitted on 13 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

From Object-Oriented Programming to Service-Oriented Computing: How to Improve Interoperability by Preserving Subtyping *

Diana Allam, Hervé Grall and Jean-Claude Royer
ASCOLA group; EMN-INRIA, LINA
Département Informatique, École des Mines de Nantes, Nantes, France
{*first_name.last_name*}*@emn.fr*

Keywords: Service-Oriented Architecture, Object-Oriented Programming, Interoperability, Loose Coupling, Subtyping

Abstract: The object-oriented paradigm is increasingly used in the implementation and the use of web services. However, the mismatch between objects and document structures in the wire has a negative impact over interoperability, more particularly when subtyping is involved. In this paper, we discuss how to improve interoperability in this context by preserving the subsumption property associated to subtyping. First we show the weaknesses of existing web service frameworks used for serialization and deserialization. Second we propose new foundations for serialization and deserialization, which leads to the specification of a new data binding between objects and document structures, compatible with subtyping.

1 Introduction

Web services are message-oriented systems based on document exchanges. But the development of web services on both client and server sides is increasingly based on object-oriented implementations. This raises the problem of impedance mismatch between objects and tree structures in XML or JSON (Lämmel and Meijer, 2007): it is difficult to preserve object properties after serialization and deserialization. This is due to the differences in the data models and the type systems between the object-oriented paradigm and the structural one. Such problems are currently hidden because of restrictions imposed by the current practice for the implementation of web services. According to the current practice, interoperability is achieved by twining the required interface at the client with the provided one at the server. In that way, web services involve a tightly coupled architecture rather than a loosely coupled one, which contradicts the standard approach for Service Oriented Architecture (SOA).

Among the restrictions imposed in the current practice for web services, we find subtyping, which is useful for web service interoperability as noticed for instance by (Carpineti and Laneve, 2006), (Lee and Cheung, 2010) and (Kourtesis and Paraskakis, 2009). Currently, object-oriented frameworks fail to preserve subtyping in its object definition. However it is possible to remove this gap while considering current techniques for serialization and deserialization: an extra adaptation on the client side is then required to make the subtype and the supertype compatible. Thus, this solution is not consistent either with the principle of loose coupling: the client and the server cannot be considered as black boxes.

In order to preserve subtyping while being conform to the loose coupling principle, we propose new foundations for serialization and deserialization, based on commutative diagrams as found in category theory (Pierce, 1991). The implementation will give rise to a new data binding framework for the two most common technologies for web services, RESTful and SOAP.

This paper is structured as follows: in Sec. 2, we present a general view of interoperability between a client and a server in existing object-oriented frameworks for RESTful and SOAP Web services. In Sec. 3, we present two problems related to subtyping in the case of a dynamic switching between services. In Sec. 4, we formalize the serialization and deserialization processes and provide their specification with commutative diagrams. In Sec. 5 we conclude this paper by proposing a solution to the subtyping problems in the formal framework proposed and by stating the position we will defend in future work.

*This work has been supported by the CESSA ANR project (see `http://cessa.gforge.inria.fr`).

2 Interoperability in object-oriented frameworks for web services

In this section we show how the web service interoperability between a client and a server is ensured in object-oriented frameworks and how it can be improved with subtyping.

A view on document exchanges. Based on our observations on some frameworks and standards, like JAX-WS, JAX-RS and Restlet, we deduce a general common exchange process of messages for existing object-oriented frameworks for web services. At each side, either the client side or the server one, messages at emission and reception must cross four levels of message types. These four levels link the object-oriented environment to the serial environment in the wire. Starting from the object-oriented level, we distinguish four levels:

1. *Data* level which refers to the input and output types of the service operations declared at the client (in the proxy interface) or at the server (in the service interface)
2. *Marshallable* level which refers to the reification format of *Data*. The marshallable message format is the essential message transformation which makes the difference between the two web service models: SOAP and RESTful. As SOAP is an activity-oriented model, the marshallable message corresponds to an activity. Thus, two types are associated to each declared service operation. The first type is associated to the request on the operation, often named with the operation name, and has as attributes the input parameters of the operation. The other type is associated to the reply which contains an attribute having the return type of the operation. For the RESTful model, which is resource oriented, transmitted messages must be resources. Here, marshallable types are equal to *Data* types.
3. *Serializable* level which refers to the type of messages ready to be serialized. These types corresponds to tree types associated to the marshallable types. In web service development practices, they are generated from the serialization annotations added into the marshallable types.
4. *Serial* level which refers to the message format ready to be sent in the wire. Serial messages are often represented in XML or JSON format.

Message transformations across these four levels at the client side is illustrated in Figure 1. These four levels appear in a symmetric way at the server side. This figure shows the different conversions applied on a message due to a function call $f(b)$, where f has type $A \rightarrow R$ and b has type B, a subtype of A. The figure distinguishes the SOAP and RESTful cases at the marshallable level as we mentioned before. For RESTful, the marshallable message is the data b at emission and a data r, an instance of R, at reception. At emission in SOAP, the marshallable message is b', instance of f_{in} where f_{in} is a reification format for function calls $f(b)$. At reception in SOAP, the marshallable message is r', an instance of f_{out} where f_{out} is a reification format for function results r.

Twining development interfaces. Building a serial message from a data at emission then building a data from the serial message at reception must lead to an equivalent instance. Therefore, an interoperability is required between the exchange levels at the client side and at the server side, as it is defined in Figure 1. According to the current practices of web service implementations, this interoperability is achieved by tightly coupling the client and the server. The object-oriented environment is associated to a structural interface expressed with Web Service Description Language (WSDL) or Web Application Description Language (WADL). Clients calling a web service must get this structural service interface and generate locally the corresponding types. Therefore, the declared operations at the client and the server are equal. Consequently, for the request and the response, the four levels for messages at the client side and the server side respectively exactly match, defining each time a serialization process and an inverse deserialization process.

Interoperability and subtyping. From a type-theoretical point of view, the equality between declared operations at the client and the server is not required to ensure a correct interoperability. The provided operation by a web service could be a subtype of the required one by the client, as recalled for instance by (Seco and Caires, 2000). Hence, according to the variance properties for subtyping, the argument type of the required operation must be a subtype of the argument type of the corresponding provided operation; and the return type of the required operation must be a supertype of the return type of the provided operation.

In the following section, we aim at showing weaknesses in existing frameworks for serialization and deserialization: they fail to preserve the subsumption property associated to subtyping when they respect the loose coupling principle.

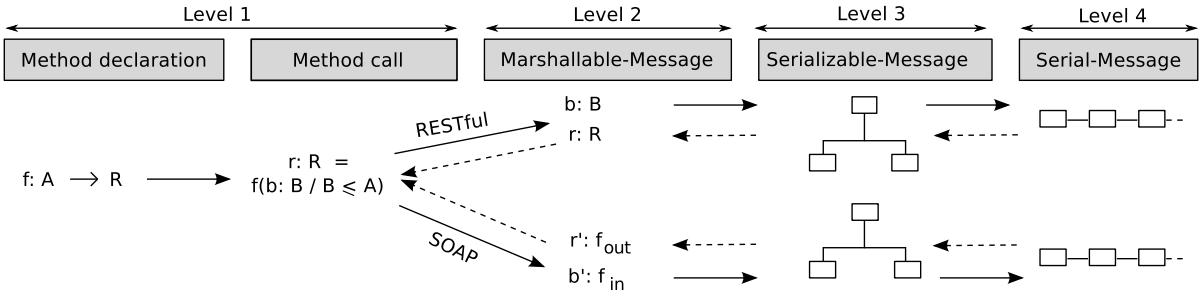

Figure 1: Message transformations from object instances to serial messages and inversely at the client side

3 Subtyping problems

In order to test the preservation of subtyping while respecting the loose coupling principle, we propose a scenario where a client can switch between two web services related by subtyping. We assume that the client is initially configured according to a *Web Service 1* interface. Then, this client would switch to a *Web Service 2*, a compatible substitution of *Web Service 1*. Switching between services having similar roles is more and more used in dynamic environments as in (Srirama et al., 2006) and (Zaplata et al., 2009). The ideal case would be by doing such a switch automatically without involving any modification to the system because a web service is replaced by a compatible one. That way, we ensure a perfect loose coupling because no modifications are required to ensure a correct interoperability. However, this ideal solution is out of the scope of existing frameworks for serialization and deserialization. In the following we will show the interoperability weaknesses in this context by considering two subtyping examples. The first example considers subtyping on provided and required operations. The second one considers subtyping between *Data* types used as input arguments.

The tests are done using Apache CXF framework [2] and JAXB Data Binding (the default one in CXF), which are popular standards and can be considered as paradigmatic. These tests consist to verify if an emitted instance could be rebuilt at reception as an instance of a supertype. The preservation of the dynamic binding (proper to object-oriented languages) from the client side to the server side is out of our interest for these tests. In order to study interoperability in the two web service models, SOAP and RESTful, we consider examples that can be deployed with both models.

Subtyping between provided and required operations. Figure 2 illustrates a view of the first example

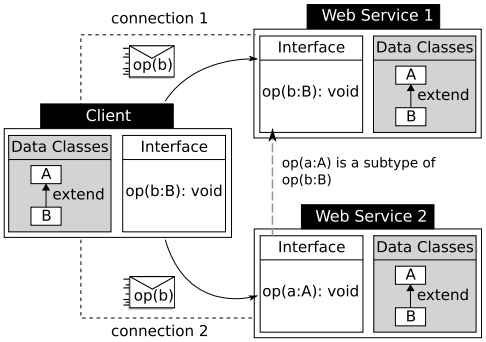

Figure 2: Interoperability and subtyping between required/provided operations

in SOAP and RESTful. Here, the operation provided by *Web Service 2* can replace the one provided by *Web Service 1* because the input type B is a subtype of the input type A, according to the object-oriented definition. The client always sends B instances when calling the *op* service. An overview of the tests done between *Client* and *Web Service 2* is given in Table 1. In this table, we note $B \equiv A$ if B has exactly the same attributes as A else it is noted $B \not\equiv A$.

Table 1 shows that interoperability works perfectly for RESTful while it works partially for SOAP.

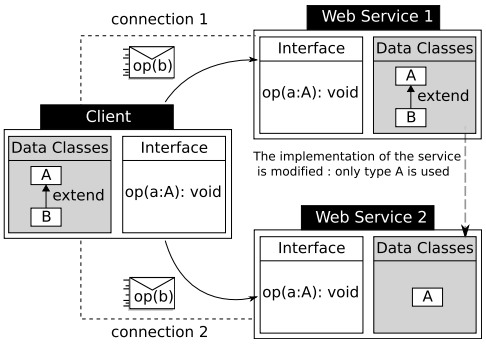

Figure 3: Interoperability and subtyping over *Data Types*

[2] http://cxf.apache.org/

		SOAP	REST
Interoperability is preserved	B $\equiv$ A	**Yes**	**Yes**
	B $\not\equiv$ A	No	**Yes**

Table 1: Results of tests on cxf framework for the example of Figure 2

			SOAP	REST
Interoperability is preserved	B is an external type	B $\equiv$ A	**Yes**	No
		B $\not\equiv$ A	No	No
	B is an internal type		**Yes**	No

Table 2: Results of test on cxf framework for the example of Figure 3

Subtyping between *Data* types. Figure 3 illustrates a view of the second example in SOAP and RESTful. Here, *Web Service 2* can replace *Web Service 1* because it declares exactly the same service. There is only one difference: *Web Service 2* does not know subtype *B*, which is an implementation detail. We consider that the client sends *B* instances when calling the *op* service operation. Table 2 shows an overview of the tests done between *Client* and *Web Service 2*. In this table, a type is called *external* when its instances can be exposed to the external world. Thus, they can be serialized according to specific serialization annotations for this type. Otherwise, the type is called *internal*. In that case, if an instance of such a type is sent to the external world, we consider the serialization annotation of its external supertype.

Table 2 shows that interoperability fails for RESTful while it works partially for SOAP.

4 Interoperability formalization

In order to fix the subtyping issues previously presented, in this section we aim at giving a formalization of the problem by referring to commutative diagrams. The message exchange mechanism of Figure 1 can be summarized as follows: in object-oriented frameworks for web services, an emitted message is an instance of a marshallable type. It will be transformed to a serializable message which is then serialized into a serial message to be transmitted in the wire. At reception, the serial message is deserialized into a serializable message which is transformed into an instance of the expected marshallable type. This mechanism can be formalized in a diagram by defining two categories, a functor and some transformations in order to go from one category to another:

1. **Category of marshallable types:** this category contains marshallable types as points and typed functions as arrows,
2. **Category of serializable types:** this category contains serializable types as points and typed functions as arrows,
3. **Binding functor:** a mapping from marshallable types to serializable types, extended to arrows,
4. **Transformations:** converting a message from a marshallable type to a serializable type is achieved by a serialization transformation. Inversely, converting a message from a serializable type to a marshallable type is achieved by a deserialization transformation.

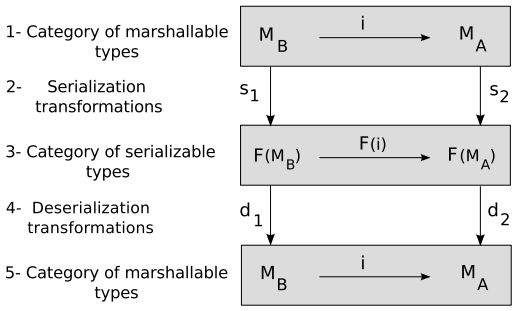

Figure 4: Fundamental Diagram for Serialization and Deserialization

In such a formalization, we consider that it is useless to specify a category for serial types because serializable messages essentially are trees, which are fully serializable: by using a depth-first traversal and a parenthesis notation, we can convert a tree into a serial data and by using the inverse function, come back to the original tree.

In Figure 4, we illustrate our formalization by considering that an instance of type B must be sent to a destination waiting an instance of supertype A. Two marshallable types, M_A and M_B, are associated respectively to A and B according to the definition given in Figure 1. These two marshallable types are related by an embedding i which allows to convert an instance of type M_B into an instance of type M_A, due to the subtyping relation between A and B. The binding functor F allows two serializable types, $F(M_B)$ and $F(M_A)$, to be associated to M_B and M_A respectively. These two serializable types are related by the

corresponding embedding $F(i)$ that allows to convert any instance in $F(M_B)$ into $F(M_A)$. In order to convert messages between categories, two serialization transformations, s_1 and s_2, transform the instances of M_B and M_A to their respective serializable forms in $F(M_B)$ and $F(M_A)$. Likewise, two deserialization transformations, d_1 and d_2, transform the instances of $F(M_B)$ and $F(M_A)$ to their respective marshallable forms in M_B and M_A.

In a diagram like this, what properties can we assume? First, it is natural to specify that the composition of the serialization transformation with the deserialization transformation produces the identity transformation, modulo a natural equivalence over marshallable types asserting that two objects are equivalent if their serialization is equal. Second, the diagram in Figure 4 may be commutative: all paths with the same source and target define by composition the same function, modulo the natural equivalence over marshallable types.

Actually, these two assumptions are fundamental to solve the subtyping issues previously described as we will discuss in the next section.

5 Discussion and future work

In both examples described in Figures 2 and 3, the whole process is described by finding a path joining M_B to M_A by crossing the category of serializable types. Actually, the path that causes problems is: $M_B \rightarrow F(M_B) \rightarrow F(M_A) \rightarrow M_A$. The trouble comes from an erroneous conversion function from $F(M_B)$ to $F(M_A)$, namely $F(i) : F(M_B) \rightarrow F(M_A)$. It is considered as equal to the identity function, which is a violation of the commutativity property. With this value, the composition of the different functions over the path is not well-defined in some cases, which generates the errors observed. But with the commutativity assumption, the problem can be tackled. Indeed, the conversion function can be easily computed:

$$F(i) = d_1; i; s_2.$$

It is obtained by composing the deserialization transformation with the embedding function and finally the serialization transformation.

Practically, $F(i)$ corresponds to a projection and possibly a renaming for the serializable structures. For instance, assume that type A has one serializable attribute x, while its subtype B has another serializable attribute y. An instance of $F(M_B)$ could be represented by the structure $B(x(val_x), y(val_y))$ (using a parenthesis notation) while an instance of $F(M_A)$ could be represented by the structure $A(x(val_x))$. Then the conversion function $F(i)$ renames the message root element from B to A and projects the couple of values (x, y) into x. Finally, the conversion function $F(i)$ can be integrated in the framework for serialization and deserialization at reception on both client and server sides, in a fully transparent way, which satisfies the loose coupling principle: this solution does not require any extra adaptation on the client side when a switch is done between compatible services.

Finally, it turns out that the diagram, when assumed commutative, could become a well-founded specification for serialization and deserialization. Its implementation could lead to the definition of a new data binding between marshallable types and serializable types, preserving the subsumption property associated to subtyping, and therefore improving interoperability.

REFERENCES

Carpineti, S. and Laneve, C. (2006). A basic contract language for web services. In *Programming Languages and Systems, 15th European Symposium on Programming (ESOP), volume 3924 of LNCS*, pages 197–213. Springer.

Kourtesis, D. and Paraskakis, I. (2009). *Semantic Enterprise Application Integration for Business Processes: Service-Oriented Frameworks*, chapter IV. Business Science Reference, Hersley.

Lämmel, R. and Meijer, E. (2007). Revealing the x/o impedance mismatch: changing lead into gold. In *Proceedings of the 2006 international conference on Datatype-generic programming*, SSDGP'06, pages 285–367, Berlin, Heidelberg. Springer-Verlag.

Lee, T. Y. L. and Cheung, D. W. (2010). Formal models and algorithms for XML data interoperability. *JCSE*, 4(4):313–349.

Pierce, B. C. (1991). *Basic Category Theory for Computer Scientists (Foundations of Computing)*. The MIT Press, 1 edition.

Seco, J. C. and Caires, L. (2000). A basic model of typed components. In *ECOOP*, pages 108–128.

Srirama, S. N., Jarke, M., and Prinz, W. (2006). Mobile web service provisioning. *Advanced International Conference on Telecommunications / Internet and Web Applications and Services, International Conference on*, 0:120.

Zaplata, S., Dreiling, V., and Lamersdorf, W. (2009). Realizing mobile web services for dynamic applications. In Godart, C., Gronau, N., Sharma, S., and Canals, G., editors, *Proceedings of the 9th IFIP Conference on e-Business, e-Services, and e-Society (I3E 2009)*, pages 240–254. Springer.