The Substitution Principle in an Object-Oriented Framework for Web Services: From Failure to Success

Diana Allam, Hervé Grall and Jean-Claude Royer
ASCOLA, Mines de Nantes - INRIA
4, rue Alfred Kastler, 44307, Nantes, France
FirstName.LastName@mines-nantes.fr

ABSTRACT

Nowadays, services are more and more implemented by using object-oriented frameworks. In this context, two properties could be particularly required in the specification of these frameworks: (i) a loose coupling between the service layer and the object layer, allowing evolutions of the service layer with a minimal impact on the object layer, (ii) an interoperability induced by the substitution principle associated to subtyping in the object layer, allowing to freely convert a value of a subtype into a supertype. However, experimenting with the popular `cxf` framework, we observed some undesirable coupling and interoperability issues, due to the failure of the substitution principle. Therefore we propose a new specification of the data binding process used to translate data between the object and service layers. We show that if the `cxf` framework followed the specification, then the substitution principle would be recovered, with all its advantages.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability—*Data mapping, Distributed objects*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*

General Terms

Languages, Reliability, Verification, Standardization

Keywords

Service-Oriented Computing, Object-Oriented Programming, Interoperability, Loose Coupling, Subtyping, Basics of Category Theory

1. INTRODUCTION

In a service-oriented architecture, services are organized as distributed components whose standardized interfaces can be published and discovered. This architecture has emerged with the Web to solve the problem of interoperability in heterogeneous distributed systems. Service components have interfaces expressed in standard languages (like `WSDL` or `WADL`) to ensure a loose coupling with the technology used to implement each component. Today, the implementation of services and of client applications is increasingly based on object-oriented programming languages. The objects exchanged between a client and a server are transformed into structured documents (expressed in `XML` or `JSON` for instance) to be communicated over the network. Thus, object-oriented frameworks for web services are essentially composed with two layers, an object layer built over a service layer.

In this context, two properties could be particularly required in the specification of these frameworks, because they turn out to be not only desirable but also expected.
(i) A *loose coupling* between the service layer and the object layer would allow evolutions of the service layer with a minimal impact on the object layer. It is interesting since there are two competing technologies for Web services, the `SOAP` (or `WS*`) one and the `RESTful` one. Thus, with the requirement satisfied, we could deploy the same object model with both technologies after a minimal adaptation. Requiring a loose coupling between two hierarchical layers is very common: this architectural principle can be found in many areas, for instance in Web services themselves, for interfaces and implementations, but also in client-service applications, for multiple tiers.
(ii) An *interoperability* induced by the *substitution principle* [12] would allow different substitutions to take place in the object layer: a value of a subtype could be exchanged between the client and the server where a value of a supertype is expected, or an interface provided by a server could be refined into a compatible one. The substitution principle associated to subtyping is of common use in object-oriented programming languages, improving flexibility: implementations can be freely refined whereas clients can be defined from black boxes.

But looking at the popular `cxf`[1] framework, which is an implementation of some paradigmatic standards for web services in the programming language `Java` (`JAX-RS API` for `RESTful` and `JAX-WS API` for `SOAP`), we observed that both requirements are not fully satisfied: several bugs arise when subtyping is introduced in a client-server interaction. This paper mainly aims at solving these issues in a general way by proposing a new specification of the data binding used to convert between objects and documents and by paving the way towards a concrete implementation in the `cxf` framework. Precisely, it makes the following contributions.

[1]The documentation of the frameworks, tools, languages, application programming interfaces, specifications, etc., cited in the paper can be found online.

• Section 2: we determine pertinent abstractions of the data flow and of the control flow in the `cxf` framework to explain how the framework processes data and how it drives the data binding.
• Section 3: we demonstrate that the `cxf` framework is unstable in presence of object subtyping.
• Section 4: from the previous abstractions, we propose a new specification founded on commutative diagrams of the data binding involved in the framework, and we show that the new specification solves the bug issues and can be concretely implemented in `cxf`.

Related work about distributed objects, data bindings and our method based on commutative diagrams are presented in Section 5 just before the conclusion.

2. A FRAMEWORK IN ACTION

An object-oriented framework for Web services like `cxf` contains two layers, one dedicated to objects, the other to services. Precisely, the object layer defines the interface between the user-defined object model and the service layer, while the service layer endorses the interaction via Web services. The section mainly explains how the framework drives one of its main component, the data binding, during the development phase and the execution phase.

2.1 Data Binding

An essential component of such a framework is the *data binding*, which binds types and values between both layers. In the object layer, values are (references to) objects and types are object types, classes or interfaces. In the service layer, values are structured documents and types are schemas: they combine two interesting properties, abstraction, leading to human readable data, and simplicity, leading to machine processable data, and particularly to serializable data, which is required for network communication. Concretely, since there are different languages for documents, the most known being `XML` and `JSON`, a framework like `cxf` accept different pluggable components for different data binding frameworks like `JAXB` (the default one) and `Aegis` for `XML`, or `Jettison` and `Jackson` for `JSON`.

The component for data binding binds object types and schemas representing their internal structure in a two-way mapping. The *schema compilation* produces object types from a schema while the *schema generation* produces a schema from object types. For instance, a class `A` can be bound to a schema giving not only the name of the type, `A`, but also its structure consisting in a sequence of field declarations. Associated to this type mapping, two functions realize conversions in a type-safe way: the *marshalling* function maps objects to documents while the *unmarshalling* function maps documents to objects. For instance, an instance of class `A` is marshalled into a document labeled with name `A` and containing a marshalling of each field as a subdocument. Figure 1 describes the different functions implemented in a data binding. Note that the data binding is restricted to specific object types, the *marshallable* types. An object type is *marshallable* if it satisfies some constraints (about its constructors and fields) and defines specific mappings to its corresponding schema. These mappings are called a *binding schema* in a reference book for `JAXB` [13]. They are described with some annotations added to the object type as in `JAXB` or with a separated binding definition as in `Aegis`. In the following, we implicitly assume that a type is marshallable, if needed.

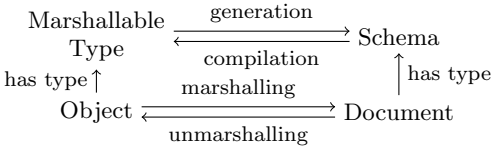


Figure 1: Data Binding

The two pairs of functions, at the level of types and values, respectively, are often presented as pairs of inverse functions. Formally, this is not the case. First, there is an impedance mismatch between schemas and object types [11], essentially due to the fact that the languages used to define schemas are too expressive. But even if we restrict ourselves to schemas generated from marshallable types, there is no biunivocal correspondence. Indeed, given a marshallable type, the binding schema could map some attributes of the class and not all of them. Therefore, the schema generation produces a schema only describing the structure of the mapped attributes; then the schema compilation produces an object type which differs from the initial one: some attributes are lacking. Hence, the non-inversibility comes from the fact that the schema generation really defines a procedure to observe objects, and this observation is partial: it accounts for only a part of the state of the object observed and it does not account for all the methods encapsulated in the object. Likewise, a marshalling followed with an unmarshalling does not preserve the object. However, in the reverse direction, we have observed in some data bindings that the following property is satisfied, although not formally specified: starting from a schema generated from marshallable types, a schema compilation followed with a schema generation preserves the schema, likewise an unmarshalling followed with a marshalling preserves a given document.

These properties induce a specific notion of equivalence over objects and object types respectively: it is the notion that we will use in the following. Two marshallable objects are equivalent if the marshalling function applied to them gives the same document, while two marshallable types are equivalent if the schema generator applied to them gives the same schema. For instance, with the default data binding `JAXB`, type `B` is equivalent to its supertype `A` when class `B` does not change the binding schema inherited from `A` and does not extend it with extra field annotations.

2.2 Development and Execution

The framework drives the data binding during the development and the execution. Typically, a development follows a process in two phases, located at the server and the client respectively.
1. Server side – Code-first approach
1.1. The developer provides some Java code to implement the service represented as an interface.
1.2. By using the object layer of the framework and especially the schema generator of the data binding, the developer generates the contract associated to the service, which specifies not only the type of the operations belonging to the service but also the port used to communicate. The contract is expressed in a dedicated language, like `WSDL` or `WADL`.

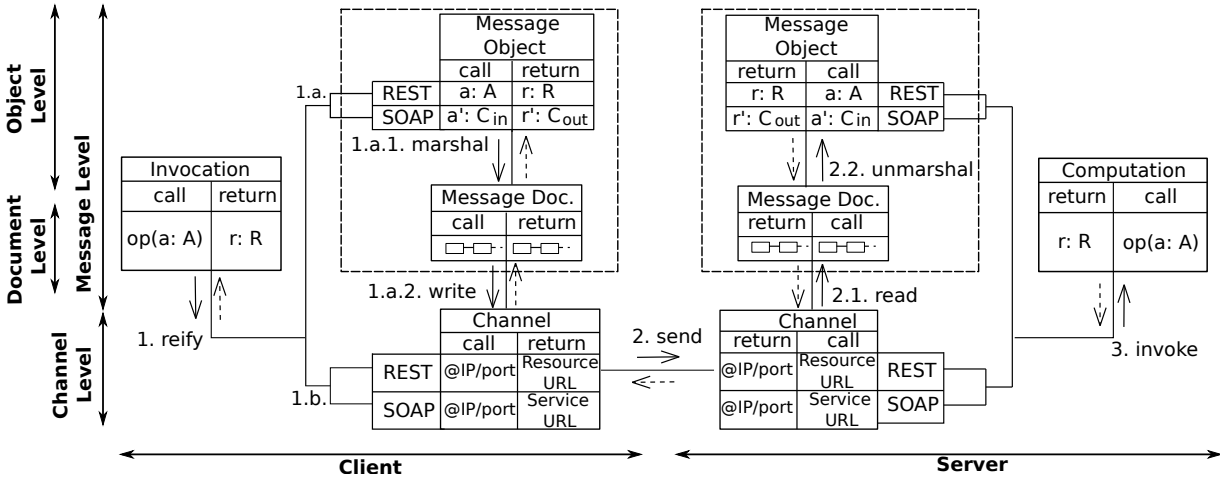


Figure 2: Data Flow in an Object-Oriented Framework for Web Services

1.3. The developer deploys the implementation and the contract on a server.
2. Client side – Contract-first approach
2.1. Using a specific tool of the object layer that embeds the schema compiler of the data binding, like in `cxf wsdl2java` or `wadl2java`, the developer produces from the contract a stub acting as a gateway towards the server and a client skeleton.
2.2. The developer completes the client skeleton to produce the client invoking the service.
The execution of a service is decomposed into an invocation on the client side and a computation on the server side in the object environment, following a request-response protocol in the service layer. Figure 2 describes the data flow involved in the execution of a service operation `R op(A a)`.
1. Client side – Invocation of the service
1.1. The client application invokes operation `op` of a service using the stub.
1.2. The invocation is reified in two components, representing a message and a channel respectively.
1.3. The framework calls the component for data binding to marshall the message.
1.4. The framework sends to the server the message marshalled over the channel, after a serialization and a binding to a concrete communication port.
2. Server side – Reception and computation
2.1. The framework receives from the client the message marshalled sent over the channel.
2.2. The framework calls the component for data binding to unmarshall the message.
2.3. From the message and the channel, the framework produces a local invocation of operation `op` of the service and calls it.
2.4. The implementation of operation `op` executes and possibly returns a result.
2.5. The framework calls the component for data binding to marshall the result and sends it to the client over the response channel.
3. Client side – Return
3.1. The framework receives from the client the response marshalled sent over the response channel.
3.2. The framework calls the component for data binding to unmarshall the response.
3.3. The invocation returns the response unmarshalled to the client application.

The main flow in Figure 2 is the marshalling of the message object into the message document, and the corresponding unmarshalling. The data flow is essentially the same, in `SOAP` and in `RESTful`. The difference between both technologies lies in the way an invocation of an operation is reified into a message and a channel: the decomposition differs. Note that in the following we omit the return flow, which leads to similar analyses and results.
`SOAP` **Case.** (i) The message contains the arguments of the operation, but also a description of the operation. Thus, as shown in Figure 2, the object message resulting from call `op(a)` is `a'`, an instance of a type $\texttt{C}_{\texttt{in}}$ representing commands associated to calls to `op` and having as attributes the input parameters of the operation. (ii) The channel identifies the target port for the whole service.
`RESTful` **Case.** (i) The message only contains the arguments of the operation. Thus, as shown in Figure 2, the object message resulting from call `op(a)` is simply `a`, instance of the input type `A`. (ii) The channel identifies not only the service but also the operation as a resource and an `http` method.

In the `cxf` framework, the data flow is implemented through a chain of interceptors. The chain is organized into a sequence of phases. In our description, we limit ourselves to the two main phases: (i) the dispatch phase used to decompose and recompose the invocation and (ii) the translation phase used to marshall and unmarshall the message. Actually, other phases are common, for instance for security. However, we can consider that they occur in the communication phase between the two main phases: they are neutral with respect to the main phases.

2.3 Driving the data binding

From the previous sections, we deduce that the framework drives the data binding through four essential functions that we detail now.

Schema generation. The framework provides an object type,

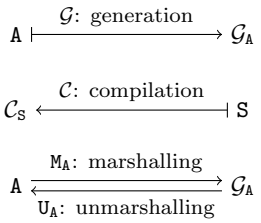


Figure 3: Driving the Data Binding

and consequently its associated tree of used types. The component for data binding returns a schema representing the structure of the observations of the object types following the defined binding schema. The schema generation is represented in Figure 3 as a function over types: function $\mathcal{G}$ maps object type `A` to schema $\mathcal{G}_A$.

Schema compilation. The framework provides a schema. The component for data binding returns a tree of object types such that their structures match the observational structures given in the binding schema. The schema compilation is represented in Figure 3 as a function over types: function $\mathcal{C}$ maps schema `S` to (root) object type $\mathcal{C}_S$.

Marshalling. The framework provides an object. The component for data binding returns a document representing an observation of the object. With the default data binding in `cxf`, the observation is based on calls to getters, so that it produces a representation, possibly partial, of the state of the object. The definition of the marshalling function is recursive: the observation of the root object may lead to objects that need to be observed, and so on. Besides the object, an extra parameter is required: indeed, the marshalling function is actually a family of functions, indexed by an object type. The family is compatible with subtyping: the observation for a subtype extends the one for a supertype. The type passed as argument by the framework to the marshalling function is bound: it lies between the dynamic type of the object, determined in `Java` by a call to method `getClass`, and its static type, coming from the declaration of the object marshalled. The bounds correspond to a dynamic selection of the marshalling function and to a static selection respectively. The marshalling function is denoted as a family of functions M_A indexed over object type `A` and defined from object type `A` to schema $\mathcal{G}_A$, as represented in Figure 3.

Unmarshalling. The framework provides a document. The component for data binding returns an object such that its marshalling produces the document given as input. With the default data binding in `cxf`, the construction of the object starts from a call to the constructor of the object class with no argument, and then is based on calls to the setters associated to the getters used in the observation during marshalling. The document provides the arguments of the setters. Again, the definition of the unmarshalling function is recursive: the construction of an object from an observation requires the construction of objects from sub-observations. Besides the document, two extra parameters are required: an object type and the schema produced from it by the schema generator. Indeed, the unmarshalling function is actually a family of functions, indexed by an object type and a schema. The schema is used to parse and validate the input document and the object type is used to construct the returned object. Both types are determined statically, from the declaration of the service. The unmarshalling function is denoted as a family of functions U_A indexed over object type `A` and defined from schema $\mathcal{G}_A$ to object type `A`, as represented in Figure 3.

To conclude, we can now formalize the definition of equivalence previously given in an informal way.

Definition 1 (Equivalence for Marshallable Types)**.** Given two marshallable types `A` and `B`, we say that `A` is equivalent to `B`, and write $A \equiv B$, when $\mathcal{G}_A = \mathcal{G}_B$. Assume $A \equiv B$; if `a` has type `A` and `b` has type `B`, we say that `a` is equivalent to `b` when $M_A(a) = M_B(b)$.

3. INTEROPERABILITY PROBLEMS

Based on concrete scenarios implemented in the `cxf` framework, the section shows the failure of the substitution principle and ends with a precise diagnose to determine the origin of the detected errors.

3.1 Failure of the Substitution Principle

With an object-oriented framework for Web services, the development process makes the client and the service tightly coupled to each other. Indeed, the object types generated from the service contract on the client are equivalent to the object types used to generate the contract on the server (see Definition 1 for equivalence). However, from an object-oriented perspective, the tight coupling should not imply usage restrictions since the *substitution principle* [12] can be applied. In its static form, the principle amounts to the *subsumption rule*: if a value has type `B`, with `B` subtype of `A`, then the value has also type `A`. Thus when a service is called, the client application could have sent an argument of a subtype while the server could have returned a result of a subtype. Dually, the interface and the implementation class on the server could be replaced with a refining interface and its implementation. A refining interface is an interface with extra methods or with specialized methods, which consume supertypes and produce subtypes, following the well-known variance rules [5]. The question raised in the article is the following:

> is the substitution principle valid in an object-oriented framework for Web services like `cxf`?

The answer is negative for `cxf`: the validity of the substitution principle has not been required in the specification of the framework. To show the result, we resort to two examples illustrating the applications of the substitution principle described above. Since the `cxf` framework implements both standard `APIs` `JAX-WS` and `JAX-RS` for `SOAP` and `RESTful` services, we can deploy the services for both technologies, which allows to study the coupling with respect to them.

Value Substitution. Figure 4 represents the interface of a service composed of one operation (method **void** `op(A a)`) and the hierarchies of data classes, on the server side and on the client one, before and after a refinement. To simplify, we assume that the class `A` is invariant after a schema generation followed with a schema compilation to produce the image of `A` on the client side. With the default data binding `JAXB`, this

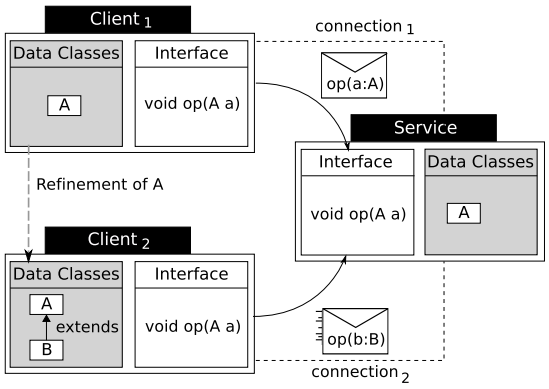


Figure 4: Value Substitution

	SOAP	RESTful
Interoperability is preserved	**Yes**	No

Table 1: Test results with `cxf` 2.5.10 for Figure 4

is the case when the class `A` only contains fields and their associated getters and setters, moreover it maps all these fields in its binding schema. After the generation of the client stub from the contract deployed on the server, the class `A` is refined into a subclass `B`. Applying the substitution principle, the client can send an instance of class `B` as argument, instead of an instance of class `A`. Testing this example in `cxf` gives different results depending on the version: `2.5.x` or `2.7.x`. The test results in `cxf 2.5.10` are presented in Table 1. This table shows some negative results: the substitution works perfectly for `SOAP` while it does not work at all for `RESTful`. In the `RESTful` case, a `javax.xml.bind.UnmarshalException` is raised with an error message stating that the structure of the received document is unexpected because it has a `B` tag as root element while the expected one should be `A`. However, for `cxf 2.7.5`, all the results are positive for both `SOAP` and `RESTful`. The migration guide between these two versions does not mention the modifications done and the rationale behind them.

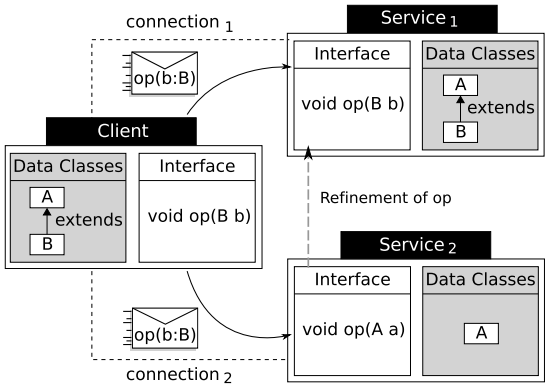


Figure 5: Interface Substitution

Interface Substitution. Figure 5 represents two services and a unique client. The client is initially configured to call $\texttt{Service}_1$. The service $\texttt{Service}_1$ is then replaced with another one, $\texttt{Service}_2$, providing the same operation `op`, but with a refined type. Precisely the operation consumes a supertype `A` of the initial type `B` while producing nothing: by application of the contravariance rule, the new type refines the first one. Again, to simplify, we assume that the classes `A` and `B` are invariant after a schema generation followed with a schema compilation to produce their images on the client side. The test results between the client and $\texttt{Service}_2$, presented in Table 2, show some negative results: interoperability works partially for `SOAP` while it does not work at all for `RESTful`. Contrary to the previous example, the results are the same for the `cxf` versions considered. In the `SOAP` case when `B` is not equivalent to `A` (`B` can map additional fields than `A` in its binding schema), the server application throws an exception of type `javax.xml.ws.soap.SOAPFaultException`. This error message refers to an unexpected additional element in the document when it is validated against the schema associated to class `A`. For the `RESTful` case, the exceptions thrown are the same as for the previous scenario based on a value substitution.

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

Table 2: Test results with `cxf` for Figure 5

3.2 Diagnosis

The two previous scenarios clearly show the failure of the substitution principle in an object-oriented framework for Web services like `cxf`. We deduce that the desirable and expected requirements are not satisfied.

(i) *Lack of interoperability*: the client and server sides are more tightly coupled than they could be. Indeed, in some common cases, the interaction between the client and the server cannot benefit from the flexibility induced by the substitution principle.

(ii) *Strong coupling*: the object and service layers are strongly coupled in `cxf`. Indeed, in some common cases, it is impossible to evolve from one service technology to another, for instance from `SOAP` to `RESTful`, without an uneasy adaptation since it involves bug corrections.

If when implementing the previous scenarios, we have detected errors, we can ask wether they really correspond to failures. The reading of the documentation, especially the developer's guide for `cxf`, and of the specification of the standards implemented, and the instability of the behavior that we have observed from one version to another lead us to the following conclusion: the validity of the substitution principle has not been required in the specification of the `cxf` framework. In other words, the errors detected do not correspond to failures, since the behavior that would be expected if the substitution principle was applied is not specified. In the following, our aim is to make the framework `cxf` adhere to the substitution principle.

We start by elaborating a diagnosis: assuming the substitution principle valid, what are the faults behind the errors detected? Then we study ad-hoc solutions, which can be developed by a user without modifying the framework. We conclude by showing that they are inappropriate.

For the scenario about the value substitution, thanks to the evolution between the versions studied, the diagnosis is easy. In the `RESTful` case for the `cxf` version `2.5.10`, the error comes from the fact that the framework calls the marshalling function at emission with the dynamic type of the object (class `B`), while the framework calls the unmarshalling function at reception with the static object type (class `A`). The error is due to the non-equivalence between both types (see Definition 1). In the `RESTful` case for the `cxf` version `2.7.5` and in the `SOAP` case, the static type is used instead for marshalling and unmarshalling. In other words, when an instance of subclass `B` is marshalled as an instance of superclass `A`, there is no error. For the scenario about the interface substitution, the fault comes from the validation of the input document, conforming to the schema associated to the subtype, against the schema associated to the supertype. When the schema differs (always in the `RESTful` case, when `A` and `B` are not equivalent in the `SOAP` case), an errors occurs.

With the default data binding in `cxf`, it is possible to force a subtype and a supertype to be equivalent with respect to a schema generation and a marshalling by making their binding schema equal. Thus, for both scenarios, an ad-hoc solution can be designed. However, the solution is not universal, since it entails a dependence over the uses of the class. Indeed, the equivalence enforcement is not contextualized: it becomes impossible to get a dedicated schema or marshalling for the subtype in other contexts where the subtype is expected with its specific features. Thus an ad-hoc solution turns out to be inappropriate. In the next section, we show how to modify the `cxf` framework in order to satisfy the substitution principle. The solution becomes fully transparent for the developer.

4. DATA BINDING: NEW SPECIFICATION

We now revisit both scenarios, studied in Section 3, while generalizing them, and propose new requirements for a data binding: the aim is to recover the validity of the substitution principle.

To express the requirements, we mainly use diagrams, which are graphs with vertices representing types and arrows representing typed functions. They represent an abstraction of the data flow described in Figure 2: an execution is described as a path between types, corresponding to a sequence of transformations of the data belonging to the types. Thus, they allow any application of the subsumption rule to be represented as a conversion function. For instance, given object types `A` and `B`, with `B` subtype of `A`, if an instance of `B` is converted into `A` during the execution, then we can represent the conversion with the following diagram:

$$\mathtt{B} \xrightarrow{\mathtt{i}} \mathtt{A},$$

where `i` is the canonical conversion function, defined in `Java` as `A i(B x) {return x;}`. A property is stated by asserting that some diagram is *commutative*: all paths with the same source and the same target are equal, as functions.

The requirements are split into *core requirements* that formalize requirements quite already satisfied in data bindings and that do not involve subtyping, and *new requirements* that involve subtyping and allow the substitution principle to be validated. The new requirements that we propose have two concrete objectives: (i) avoiding all the faults analyzed in Section 3.2, (ii) ensuring that any object at emission is equivalent to the corresponding object at reception. In a diagram, an error caused by a fault is represented in the function where it happens as a dotted arrow.

$$\mathtt{B} \dashrightarrow \mathtt{A}$$

As for the equivalence, it can be represented as a specific commutativity property between paths in each case, as we will see.

4.1 From Diagrams to Requirements

To determine the different requirements, we need to separately study the two cases `SOAP` and `RESTful`. We start by the simplest case, the `RESTful` one.

The RESTful case. Consider the initial situation in the first scenario, when a client sends an instance of type `A` to the server providing an operation **void** `op(A a)`: no subtyping is involved. The development and execution processes can be pictured as follows.

$$\mathcal{CG}_{\mathtt{A}} \xrightarrow{\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}} \mathcal{G}_{\mathtt{A}} \xrightarrow{\mathtt{U}_{\mathtt{A}}} \mathtt{A}$$

The development process allows the sequence of types to be built, from right to left: applied to `A`, the schema generation produces schema $\mathcal{G}_{\mathtt{A}}$ and then the schema compilation produces object type $\mathcal{CG}_{\mathtt{A}}$. Note that here and in the following, we consider the general case, when the object types `A` and $\mathcal{CG}_{\mathtt{A}}$ are not assumed to be equal: in Section 3, to simplify we have assumed that they were equal. The execution process allows the sequence of transformations (arrows) to be built, from left to right. The marshalling function $\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}$ transforms any instance of type $\mathcal{CG}_{\mathtt{A}}$ into a document conforming to schema $\mathcal{G}_{\mathtt{A}}$ for transmission. After reception, the unmarshalling function $\mathtt{U}_{\mathtt{A}}$ transforms the document into an object of type `A`. As the marshalling function $\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}$ has type $\mathcal{CG}_{\mathtt{A}} \to \mathcal{GCG}_{\mathtt{A}}$, we need the following property to ensure that the whole transformation is well-typed.

Core Requirement 1 (Compilation Inversibility)**.** *The schema generation is a retraction (left inverse) of the schema compilation, when restricted to schemas resulting from the schema generation.*

$$\forall\, \mathtt{A}\,.\ \mathcal{GCG}_{\mathtt{A}} = \mathcal{G}_{\mathtt{A}}.$$

The equivalence between the initial object and the final one can be represented with the following diagram, by asserting that it is commutative.

$$\mathcal{CG}_{\mathtt{A}} \xrightarrow{\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}} \mathcal{G}_{\mathtt{A}} \xrightarrow{\mathtt{U}_{\mathtt{A}}} \mathtt{A} \xrightarrow{\mathtt{M}_{\mathtt{A}}} \mathcal{G}_{\mathtt{A}}, \qquad \mathcal{CG}_{\mathtt{A}} \xrightarrow[\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}]{} \mathcal{G}_{\mathtt{A}}$$

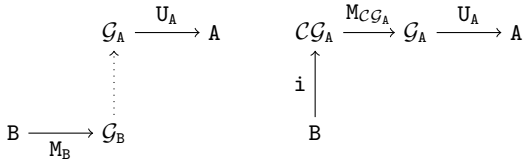
In other words, the objects are equivalent if their marshallings are equal, which leads to the following sufficient property, completing the previous requirement.

Core Requirement 2 (Unmarshalling Inversibility)**.** *The marshalling function is a retraction of the unmarshalling function, when restricted to schemas resulting from the schema generation.*

$$\forall\, \mathtt{A}\,.\; \mathtt{U_A} \,;\, \mathtt{M_A} = \mathtt{id}_{\mathcal{G}_\mathtt{A}}.$$

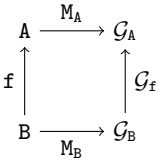
We now consider requirements dealing with subtyping. Our diagrams are enriched with another row: the bottom row deals with subtypes while the top row deals with supertypes.

Revisit the first scenario involving a value substitution (see Figure 4): in its simplified form, a client sends an instance of subtype `B` while the server expects an instance of supertype `A`. After generalization, we get the following diagrams, corresponding for the left hand side to the older version of `cxf` and for the right hand side to the more recent one.



The object type `B` is a subtype of $\mathcal{CG}_\mathtt{A}$, as witnessed by the canonical conversion function `i` in the diagram on the right. The diagram on the left shows an error: the framework calls the marshalling function with the dynamic type `B`, producing a document with schema $\mathcal{G}_\mathtt{B}$; this document cannot be converted into schema $\mathcal{G}_\mathtt{A}$. On the other hand, the diagram on the right shows no error: the framework calls the marshalling function with the static type, $\mathcal{CG}_\mathtt{A}$. To avoid the error, it suffices to provide the ability to convert between schemas, which leads to the following requirement.

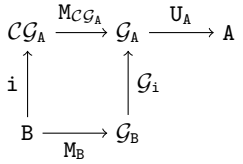
New Requirement 3 (Lifting for Schema Generation)**.** *The schema generation $\mathcal{G}$ is a functor*[2] *and the marshalling function* `M` *is a natural transformation: given any object types* `A` *and* `B`*, for any (interesting*[3]*) function* $\mathtt{f} : \mathtt{B} \to \mathtt{A}$*, there exists a function $\mathcal{G}_\mathtt{f} : \mathcal{G}_\mathtt{A} \to \mathcal{G}_\mathtt{B}$, the lifting of* `f`*, such that the following diagram commutes.*



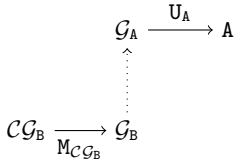
The lifting $\mathcal{G}_\mathtt{f} : \mathcal{G}_\mathtt{B} \to \mathcal{G}_\mathtt{A}$ is effectively computable, thanks to Core Requirement 2 and New Requirement 3.

$$\begin{aligned} \mathtt{M_B} \,;\, \mathcal{G}_\mathtt{f} &= \mathtt{f} \,;\, \mathtt{M_A} \\ \mathtt{U_B} \,;\, \mathtt{M_B} \,;\, \mathcal{G}_\mathtt{f} &= \mathtt{U_B} \,;\, \mathtt{f} \,;\, \mathtt{M_A} \\ \mathcal{G}_\mathtt{f} &= \mathtt{U_B} \,;\, \mathtt{f} \,;\, \mathtt{M_A} \end{aligned}$$

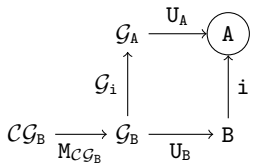
Moreover, with the new requirement, the choice between a static type or a dynamic type to be passed to the marshalling function does not matter. Indeed the following diagram is now commutative (thanks to Core Requirement 1 that gives the equality $\mathcal{GCG}_\mathtt{A} = \mathcal{G}_\mathtt{A}$).



Let us revisit the second scenario involving an interface substitution (see Figure 5): in its simplified form, a client, first connected to a server providing operation **void** `op(B b)`, sends an instance of type `B` to a new server providing operation **void** `op(A a)`, with `B` subtype of `A`. After a generalization, the execution, which produces an error, can be pictured as follows.



Again, the error comes from the impossibility to convert between schemas. Now with New Requirement 3, we can convert from $\mathcal{G}_\mathtt{B}$ to $\mathcal{G}_\mathtt{A}$: just use lifting $\mathcal{G}_\mathtt{i}$, where `i` is the canonical conversion function from subtype `B` to type `A`. If the fault is repaired, it remains to determine a sensible notion of equivalence between the initial object and the final object. Definition 1 allows objects to be compared, provided that they belong to equivalent marshallable types. But here they may be not equivalent. A first workaround should be to define the equivalence as follows: the objects are equivalent if they are transformed into the same document with schema $\mathcal{G}_\mathtt{A}$, in other words, if $\mathtt{M}_{\mathcal{CG}_\mathtt{B}} \,;\, \mathcal{G}_\mathtt{i} = \mathtt{M}_{\mathcal{CG}_\mathtt{B}} \,;\, \mathcal{G}_\mathtt{i} \,;\, \mathtt{U_A} \,;\, \mathtt{M_A}$, which can be directly deduced from Core Requirement 2. But we can also propose another definition, more accurate. Indeed, another substitution could take place. Instead of replacing operation **void** `op(B b)` with operation **void** `op`'(A a), we could just modify the implementation of operation `op` as **void** `op(B b)` {`op`'(b);}. Applying the substitution principle, we could deduce that the arguments received by `op`' should be equivalent, in other words that the following diagram should be commutative.



In this diagram, we have surrounded type `A` to mean that the equality between paths must be considered modulo object equivalence over `A`, in other words modulo a composition with the marshalling function: for two paths $\mathtt{f} : X \to \mathtt{A}$ and $\mathtt{g} : X \to \mathtt{A}$, their equivalence in the diagram means the equation $\mathtt{f} \,;\, \mathtt{M_A} = \mathtt{g} \,;\, \mathtt{M_A}$ instead of $\mathtt{f} = \mathtt{g}$. Commutativity therefore means here:

$$\mathtt{M}_{\mathcal{CG}_\mathtt{B}} \,;\, \mathcal{G}_\mathtt{i} \,;\, \mathtt{U_A} \,;\, \mathtt{M_A} = \mathtt{M}_{\mathcal{CG}_\mathtt{B}} \,;\, \mathtt{U_B} \,;\, \mathtt{i} \,;\, \mathtt{M_A},$$

which can be deduced from $\mathtt{U_A} \,;\, \mathtt{M_A} = \mathtt{id_A}$ and $\mathcal{G}_\mathtt{i} = \mathtt{U_B} \,;\, \mathtt{i} \,;\, \mathtt{M_A}$.

[2] A function defined over types is a *functor* when it can be extended to a function over typed functions.

[3] The set of interesting functions could be defined as the set of functions satisfying the commutativity property. In any case, it should contain the set of canonical conversion functions.

The SOAP case. With the `SOAP` technology, the sequence of transformations from the client to the server is complexified during a call $\mathtt{op}(\mathtt{a})$ since it involves a call reification, as recalled in Figure 2. Indeed, at emission, before the marshalling, the argument, an object `a` with type `A`, is first embedded into a *command* that has type $\mathtt{C}_{\mathtt{in}}$ and reifies the call. Then it is the command that is marshalled and sent. Symmetrically, at reception, the unmarshalling produces a command, `c`, with type $\mathtt{C}_{\mathtt{in}}$. A projection is then applied to command `c`, producing the argument. To describe the call reification, we use one function over types, corresponding to the command generation, and two transformations, indexed over object types, from object types to command types and conversely, an embedding and a projection respectively.

Command generation. $\mathcal{F}_{\mathtt{A}}$ represents the command type associated to object type `A`. We omit the dependence over the operation, which is assumed to be fixed.

Embedding. Given an object type `A`, $\mathtt{E}_{\mathtt{A}}$ represents the embedding function defined from `A` to $\mathcal{F}_{\mathtt{A}}$.

Projection. Given an object type `A`, $\mathtt{P}_{\mathtt{A}}$ represents the projection function defined from $\mathcal{F}_{\mathtt{A}}$ to `A`.

Figure 6 sums up these definitions.

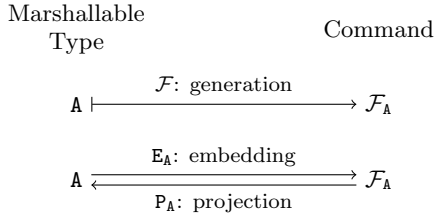


Figure 6: Command Generation

We now define the requirements for the call reification, in the same vein as before for the `RESTful` case.

For a type `A` used in a monomorphic way, the development and execution processes can be pictured as follows.

$$\mathcal{CG}_{\mathtt{A}} \xrightarrow{\mathtt{E}_{\mathcal{CG}_{\mathtt{A}}}} \mathcal{CGF}_{\mathtt{A}} \xrightarrow{\mathtt{M}_{\mathcal{CGF}_{\mathtt{A}}}} \mathcal{GF}_{\mathtt{A}} \xrightarrow{\mathtt{U}_{\mathcal{F}_{\mathtt{A}}}} \mathcal{F}_{\mathtt{A}} \xrightarrow{\mathtt{P}_{\mathtt{A}}} \mathtt{A}$$

We assume that the initial type is the same as the one for the `RESTful` case, namely $\mathcal{CG}_{\mathtt{A}}$: this constraint allows to switch between service technologies with a minimal impact on the object application (on the client side). To get the whole transformation well-defined and well-typed, we need two preliminary requirements. The first one allows the initial embedding to be well-defined.

Core Requirement 4 (Commutativity)**.** *The command generation and the composition of the schema compilation and generation commutes:*

$$\forall\, \mathtt{A}\,.\ \mathcal{CGF}_{\mathtt{A}} = \mathcal{F}_{\mathcal{CG}_{\mathtt{A}}}.$$

The second one allows the initial type to be defined in a deterministic way.

Core Requirement 5 (Injectivity of Command Generation)**.** *The command generation is injective:*

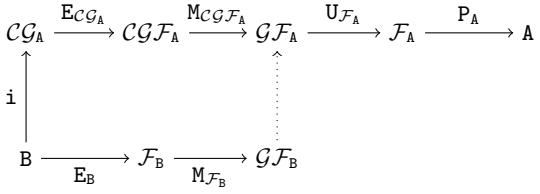
$$\forall\, \mathtt{A}\,.\ \mathcal{F}_{\mathtt{A}} = \mathcal{F}_{\mathtt{B}} \Rightarrow \mathtt{A} = \mathtt{B}.$$

Thus, the command generation has a retraction (left implicit). Then, the initial object and the final object must also be equivalent: their marshalling must be equal. To simplify, we assume the equality after an embedding: $\mathtt{E}_{\mathcal{CG}_{\mathtt{A}}}\,;\mathtt{M}_{\mathcal{CGF}_{\mathtt{A}}} = \mathtt{E}_{\mathcal{CG}_{\mathtt{A}}}\,;\mathtt{M}_{\mathcal{CGF}_{\mathtt{A}}}\,;\mathtt{U}_{\mathcal{F}_{\mathtt{A}}}\,;\mathtt{P}_{\mathtt{A}}\,;\mathtt{E}_{\mathtt{A}}\,;\mathtt{M}_{\mathcal{F}_{\mathtt{A}}}$. A sufficient condition, perhaps stronger than necessary but natural, follows.

Core Requirement 6 (Projection Inversibility)**.** *The projection and the embedding functions form a pair of inverses:*

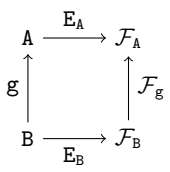
$$\forall\, \mathtt{A}\,.\ (\mathtt{P}_{\mathtt{A}}\,;\mathtt{E}_{\mathtt{A}} = \mathtt{id}_{\mathcal{F}_{\mathtt{A}}}) \wedge (\mathtt{E}_{\mathtt{A}}\,;\mathtt{P}_{\mathtt{A}} = \mathtt{id}_{\mathtt{A}}).$$

We now come to the new requirements dealing with subtyping. Consider again the first scenario, involving a value substitution (see Figure 4). The execution can be pictured as follows.



Besides the execution path starting with a conversion from `B` to $\mathcal{CG}_{\mathtt{A}}$ via `i`, we have also represented the initial path corresponding to an execution where the framework would have called the marshalling function with a dynamic type. To allow both choices, not only the static one but also the dynamic one, as for the `RESTful` case, we need the following requirement.

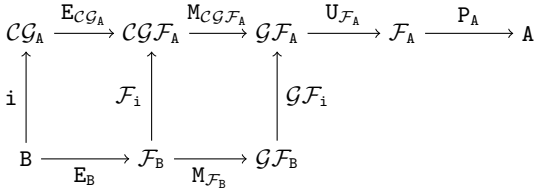
New Requirement 7 (Lifting for Command Generation)**.** *The command generation $\mathcal{F}$ is a functor and the embedding function* `E` *is a natural transformation: given any object types* `A` *and* `B`*, for any (interesting) function* $\mathtt{g} : \mathtt{B} \to \mathtt{A}$*, there exists a function $\mathcal{F}_{\mathtt{g}} : \mathcal{F}_{\mathtt{B}} \to \mathcal{F}_{\mathtt{A}}$, the lifting of* `g`*, such that the following diagram commutes.*



As for the schema generation, the lifting $\mathcal{F}_{\mathtt{g}} : \mathcal{F}_{\mathtt{B}} \to \mathcal{F}_{\mathtt{A}}$ is effectively computable:

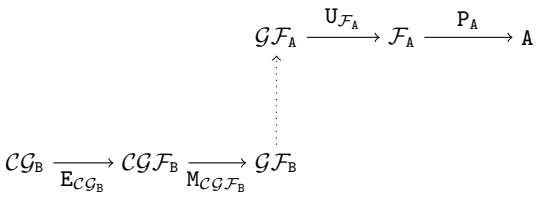
$$\mathcal{F}_{\mathtt{g}} \quad = \quad \mathtt{P}_{\mathtt{B}}\,;\mathtt{g}\,;\mathtt{E}_{\mathtt{A}}.$$

We can now deduce that the following diagram is commutative: the choice between static types and dynamic types does not matter.

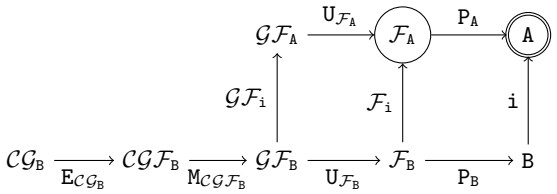


Let us consider the second scenario involving an interface substitution (see Figure 5). The execution, which produces

an error, can be pictured as follows.



Again, the error comes from the impossibility to convert between schemas. But we can use New Requirements 3 and 7 to compute a conversion as lifting $\mathcal{GF}_\mathtt{i}$. We get the following diagram.



It is commutative, provided that the commutativity of paths is interpreted for $\mathcal{F}_\mathtt{A}$ modulo marshalling and for $\mathtt{A}$ modulo embedding followed by marshalling: for two paths $\mathtt{f} : X \to \mathcal{F}_\mathtt{A}$ and $\mathtt{g} : X \to \mathcal{F}_\mathtt{A}$, their equivalence in the diagram means the equation $\mathtt{f} \,;\mathtt{M}_{\mathcal{F}_\mathtt{A}} = \mathtt{g} \,;\mathtt{M}_{\mathcal{F}_\mathtt{A}}$, and for two paths $\mathtt{f} : X \to \mathtt{A}$ and $\mathtt{g} : X \to \mathtt{A}$, their equivalence in the diagram means the equation $\mathtt{f} \,;\mathtt{E}_\mathtt{A} \,;\mathtt{M}_{\mathcal{F}_\mathtt{A}} = \mathtt{g} \,;\mathtt{E}_\mathtt{A} \,;\mathtt{M}_{\mathcal{F}_\mathtt{A}}$. Thus, the final objects that are computed are equivalent, as for the `RESTful` case.

Finally, with all the requirements defined in this section, the substitution principle can be recovered, in a robust way. All the problems presented in Section 3 come from faults in the computation of the lifting functions used to lift conversions from objects to documents. With the new requirements, we are able to precisely specify these lifting functions. If `i` is the canonical conversion from `B` to `A`, the liftings are defined as follows, in the `RESTful` case and in the `SOAP` case respectively:

$$\mathcal{G}_\mathtt{i} \quad = \quad \mathtt{U}_\mathtt{B} \,;\mathtt{i} \,;\mathtt{M}_\mathtt{A}, \tag{1}$$

$$\mathcal{GF}_\mathtt{i} \quad = \quad \mathtt{U}_{\mathcal{F}_\mathtt{B}} \,;\mathtt{P}_\mathtt{B} \,;\mathtt{i} \,;\mathtt{E}_\mathtt{A} \,;\mathtt{M}_{\mathcal{F}_\mathtt{A}}. \tag{2}$$

4.2 Towards an Implementation

We now pave the way towards an implementation in the `cxf` framework of the lifting functions used for conversion, following Equations 1 and 2. Our aim is a full integration into the framework in a transparent way for the user of the framework.

Meaning of the Specification. In `cxf`, the specification given by Equations 1 and 2 can be interpreted as follows. For the `RESTful` case, on input $\mathtt{doc}_\mathtt{B}$ with type $\mathcal{G}_\mathtt{B}$, the framework must successively call (i) the unmarshalling function by passing type `B`, (ii) the conversion function from `B` to `A`, (iii) the marshalling function by passing type `A` and get a document $\mathtt{doc}_\mathtt{A}$ with type $\mathcal{G}_\mathtt{A}$. For the `SOAP` case, on input $\mathtt{doc}_{\mathcal{F}_\mathtt{B}}$ with type $\mathcal{GF}_\mathtt{B}$, the framework must successively call (i) the unmarshalling function by passing type $\mathcal{F}_\mathtt{B}$, produced by the command generation from `B`, (ii) the projection function, corresponding to the call to a getter of command class $\mathcal{F}_\mathtt{B}$, (iii) the conversion function from `B` to `A`, (iv) the embedding function by passing type `A`, corresponding to calls to the constructor of command class $\mathcal{F}_\mathtt{A}$ and to a setter, (v) the marshalling function by passing type $\mathcal{F}_\mathtt{A}$ and get a document $\mathtt{doc}_{\mathcal{F}_\mathtt{A}}$ with type $\mathcal{GF}_\mathtt{A}$.

Lifting Algorithm. With an implementation directly following the specification, the framework needs to know both types `A` and `B`, which is a strong constraint. Moreover, the conversion is not efficient since it involves a double translation, from documents to objects and back. A better solution is to directly define a transformation from documents to documents. With the default data binding `JAXB`, using `XML`, the documents differ in two points: (i) the name of the root tag, (ii) the presence of additional elements due to the mapping of additional attributes. Hence, in order to transform document $\mathtt{doc}_\mathtt{B}$ into document $\mathtt{doc}_\mathtt{A}$ with type $\mathcal{G}_\mathtt{A}$, it suffices to rename if needed the root tag, and to extract if possible the sub-documents of $\mathtt{doc}_\mathtt{B}$ to match the definition of type $\mathcal{G}_\mathtt{A}$. This extraction corresponds to a well-studied algorithm, the *tree inclusion problem* [4]. The same conclusion applies to the `SOAP` case.

Deployment of the implementation. According to the previous diagrams describing the execution paths, the conversion due to the lifting functions occurs after the marshalling at emission and before the unmarshalling at reception. It could be implemented inside the data binding driven by the framework at emission or reception, by a modification of the marshalling function and the unmarshalling function respectively. But this solution implies modifying all the existing frameworks for data binding or defining from scratch a new data binding. As both alternatives are costly solutions, we prefer to modify the framework by adding an interceptor between the marshalling function at emission and the unmarshalling function at reception. Indeed, thanks to a flexible architecture, it is relatively easy in `cxf` to add an interceptor to the interceptor chains, composed of the outbound interceptor chain at emission and the inbound interceptor chain at reception. Since the direct lifting algorithm does not depend on type `B`, known at emission, but only on type `A`, known at reception, we can determine the best place to deploy the interceptor: it is at reception in the inbound interceptor chain.

5. RELATED WORK

The specification that we have developed for a data binding aims at validating the substitution principle for an object-oriented framework for Web services. In an introduction to subtyping in object-oriented distributed systems, Indulska [9] considers the substitution principle as a core requirement: our position is indeed that its validity is a main concern. We can ask whether the question that we have raised in the context of an object-oriented framework for Web services has been solved in other contexts: we mainly think to two other standards for interoperability between distributed object-oriented applications: Common Object Request Broker Architecture (`CORBA`) [16] and Remote Method Invocation (`RMI`) [17]. With `CORBA`, which allows heterogeneous environments, contrary to `RMI`, the substitution principle is also valid. Indeed, the interfaces, which are object-oriented, are equipped with a subtying relation that can be used to ensure substituability. With `RMI`, the substitution principle is valid, to the extent that it is valid in `Java`, since the RMI system of distributed `Java` objects follows the `Java` object model whenever possible. It appears that the main reason

that entails the failure of the substitution principle in object-oriented frameworks for Web services comes from the data binding required between the object layer and the service one.

A data binding is a method to process structured documents. There are two main alternative methods: (i) the old fashioned method, based on dedicated `API`, which are still used as back end (like `SAX` or `DOM` `API`s for `XML`), (ii) specialized programming languages directly integrating documents as data, like `CDuce` [3]. The trend towards an integration into existing programming languages predominantly (and perhaps overwhelmingly) involves functional programming languages. An essential reason stands in the proximity of the data models: terms and algebraic types versus documents and schemas. In the same trend, for object-oriented programming languages, the integration mainly lies in the extension with functional constructs [7, 6]. Likewise, smooth formalizations for data bindings can be found for functional programming languages, like `Haskell` [15, 2]. In contrast, the data bindings for object-oriented programming languages suffers from an impedance mismatch [11]. An essential reason stands in the gap between data models: observations and coalgebraic types [10] (instead of terms and algebraic types) versus documents and schemas. Starting from standards like `JAXB`, more elaborate solutions [1] have reduced the gap. These solutions consider the reverse direction than ours: from schemas to object types and from documents to objects. Moreover, to the best of our knowledge, no formal specification for a data binding is given.

Thanks to diagrams and to commutativity properties, our specification is not only formal but still intuitive. Actually, it has a solid background, category theory. Precisely, with respect to the categorical manifesto of Goguen [8], our specifications conforms to the first dogma (category for interpreting a programming language), the second one (functor for the constructions associated to a data binding) and the third one (natural transformation associated to the preceding constructions). Thus, our approach to conversion is akin to the pioneer work of Reynolds [14].

6. CONCLUSION AND FUTURE WORK

The paper shows how to recover the substitution principle [12] in an object-oriented framework for web services, namely `cxf`. We propose a specification of the data binding used to relate data and types between the object and service layers. The specification splits into two subsets, the core requirements, which are generally implicit but satisfied by current data bindings, especially by `JAXB`, the default one for `cxf`, and new requirements, dealing with subtyping and allowing the substitution principle to be recovered. The new requirements mainly specifies how to lift conversions from objects to documents. Indeed, the inability to convert documents as objects are converted by subsumption produces all the errors that we have detected when experimenting with `cxf`.

The specification is based on (commutative) diagrams, describing an abstraction of the data flow. Thanks to its intuitive interpretation, it can directly lead to an implementation. Two directions can be considered: (i) a modification of the framework, the light way that we have described and that consists in extending the driving of the plugged data binding, or (ii) a modification of the data binding, and even the creation of a new data binding. Thanks to its solid foundations, basic category theory, the specification can be fully formalized, with the associated advantages during the design and validation phases. It could lead to the definition of an abstract model for a framework and a data binding, which could then be refined into different object-oriented programming languages. This opportunity would allow different environments to interoperate, whereas the current paper is limited to the use of the same language and framework on the client side and on the server side. It does not seem to be a serious limitation.

7. REFERENCES

[1] S. Alagic and P. A. Bernstein. Mapping XSD to OO Schemas. In *ICOODB'09*, volume 5936 of *LNCS*, pages 149–166, 2009.

[2] F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema-Haskell Data Binding. In *PADL'04*, volume 3057 of *LNCS*, pages 71–85, 2004.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-Centric General-Purpose Language. *SIGPLAN Notices*, 38(9):51–63, 2003.

[4] P. Bille and I. L. Gortz. The Tree Inclusion Problem: In Linear Space and Faster. *ACM Transactions on Algorithms*, 7(3):38:1–38:47, 2011.

[5] G. Castagna. Covariance and Contravariance: Conflict without a Cause. 17(3):1–17, 1995.

[6] A. Christensen, C. Kirkegaard, and A. Møller. A Runtime System for XML Transformations in Java. In *XSym'04*, volume 3186 of *LNCS*, pages 143–157, 2004.

[7] V. Gapeyev, F. Garillot, and B. C. Pierce. Statically Typed Document Transformation: An Xtatic Experience. In *PLAN-X'06*, pages 2–13, 2006.

[8] J. Goguen. A Categorical Manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[9] J. Indulska. Subtyping in Distributed Systems. In *Formal Methods for Distributed Processing: a Survey of Object-oriented Approaches*, pages 233–253. 2001.

[10] B. Jacobs. Objects and Classes, Co-Algebraically. In *Object Orientation with Parallelism and Persistence*, pages 83–103. 1995.

[11] R. Lämmel and E. Meijer. Revealing the X/O Impedance Mismatch: Changing Lead into Gold. In *SSDGP'06*, volume 4719 of *LNCS*, pages 285–367, 2007.

[12] B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[13] B. McLaughlin. *Java and XML Data Binding*. O'Reilly, 2002.

[14] J. C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. volume 94 of *LNCS*, pages 211–258, 1980.

[15] P. Thiemann. A Typed Representation for HTML and XML Documents in Haskell. *Journal of Functional Programming*, 12(5):435–468, 2002.

[16] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[17] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290, 1996.