Loose coupling and substitution principle in objet-oriented frameworks for web services

Diana Allam

Thèse de Doctorat

Diana ALLAM

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)
Ecole des Mines de Nantes

Soutenue le 10 Juillet 2014
Thèse n° : ED 503-2014 EMNA 0115

Loose Coupling and Substitution Principle in Object-Oriented Frameworks for Web Services

JURY

Rapporteurs :	**M. Farouk TOUMANI**, Professeur, Université Blaise Pascal
	M. Gwen SALAÜN, Maître de conférences, ENSIMAG, Grenoble INP
Examinateurs :	**M. Claude JARD**, Professeur, Université de Nantes
	M. Anderson SANTANA DE OLIVEIRA, Chercheur senior, SAP Labs
Directeur de thèse :	**M. Jean-Claude ROYER**, Professeur, Ecole des Mines de Nantes
Co-directeur de thèse :	**M. Hervé GRALL**, Maître-Assistant, Ecole des Mines de Nantes

Remerciements

[1]

Je tiens à remercier dans un premier temps, mon encadrant, Hervé Grall, et mon directeur de thèse, Jean-Claude Royer, pour leurs patiences pendant nos heures de discussion, pour leurs aides, leurs conseils avisés mais également pour leurs soutiens tout au long de ma thèse. Un grand MERCI!

Merci à Gwen Salaün et Farouk Toumani pour avoir accepté d'être rapporteurs. Merci aussi à Claude Jard et Anderson Santana de Oliveira pour avoir accepté d'être membres du Jury.

Merci à l'ensemble des membres du projet ANR CESSA, Mario Südholt, Rémi Douence, Hervé Grall, Jean-Claude Royer, Matteo Dell-Amico, Yves Roudier, Muhammad Sabir Idrees, Anderson Santana de Oliveira, Gabriel Serme et Gaëtan Harel avec qui j'ai eu des précieuses échanges.

Merci au chef d'équipe Ascola, Mario Südholt, à l'ancien chef du département, Narendra Jussien et au nouveau chef du département, Jacques Noyé, d'avoir fourni un soutien financier pour participer à des divers activités et événements, particulièrement à des conferences nationales et internationales.

Merci à Catherine Fourney, Florence Rogues, Cécile Derouet, Diana Gaudin, Michelle Dauvé, Sylvie le Goff et Delphine Turlier pour leurs aides en tout ce qui concerne les étapes administratives et les demandes de mission.

Merci aux membres du département informatique pour leur belle ambiance amicale. Je tiens à remercier en particulier Jurgen pour sa disponibilité et ses conseils.

Merci aux membres de CXF mailing-list, particulièrement Sergey Beryozkin, pour leurs réponses rapides à mes questions sur le forum et leurs aides à résoudre et comprendre certains problèmes.

Merci à mes amis du département énergétique à l'école des Mines, Nadia, Charlotte, Emna, Elias, Maël et les autres, avec qui j'ai passé des très beaux moments.

[1] http://lylouannecollection.blogspot.fr/

Merci à mes amis Libanais à Nantes (et qui sont passés par Nantes), Rémy, Nader et Ali el Housseini, Ali el Roz, Ali Yassin, Jaafar, Houssam, Haydar Abbass, Ghassan, Ahmad, Mohamed Jawad, Hosni, Ayman & Batoul, Zein & Majd, Youmna et les autres avec qui j'ai partagé des beaux souvenirs.

Merci à ma famille de Roumanie, Carmen, Cilvio, Laura, Andra, Christina, Marta, Rodica, Alex, ma grande mère et les autres qui me sont chers.

Merci à ma famille ALLAM de Hermel, particulièrement à mes cousins Dyala, Faten, Firas, Yahya, Zakaria, Mohamed, Malek, Sam, Khouzama, Malak, Rami, Ali, Layal, Rasha, Bachir, Houssein, Fatima[2], Emne, Jawad, Widiane, Samaher, Amani, Lamis et tous les autres.

Merci à la petite famille ASSAF, particulièrement à Mouchira, Sara et Nour, pour les beaux moments qu'on a passé ensemble et les beaux souvenirs inoubliables.

Merci à mes deux plus chères personnes, mon père et ma mère, à qui je dois plus que toutes leurs années de sacrifice, leurs soutiens sans faille, leurs disponibiltés à tout moment, leurs encouragements et leurs prières. C'est grâce à vous deux que je suis qui je suis aujourd'hui.

Merci à mes deux soeurs, Nadine et Karmen, et à mon frère Karim, à qui je réserve une place très particulière dans mon coeur et dans ma vie.

Merci à mon précieux, Ali, mon compagnon du chemin, l'étoile qui éclaire ma vie et mon ange protecteur.

Contents

1

Introduction

Web services are important today as they are part of our daily life: to share photos with friends using *Flickr*, to buy products using *eBay* or to pay online using *PayPal*, we use Web services. The providers of these applications publish their services as API interfaces. For instance:

- *Flickr* API[1] can be used to retrieve photos from the Flickr photo sharing service using a variety of public photos and videos, favorites, friends, group pools, discussions, and more.
- *eBay* API[2] allows developers to list items, manage user information, get item information, and manage eBay sales and purchases.
- *PayPal* API[3] offers online payment solutions and has more than 153 million customers worldwide.

These APIs interfaces are reachable through two main protocols (or models): `SOAP` which is an activity based model (RPC style) and `RESTful` which is a resource based model (Web style) [38]. The difference between these two models could be easily represented by the following example. For a flight reservation service, a client first searches a flight travel giving the source city, the destination city and the date, then books the most convenient travel. In `SOAP`, this service is represented as two activities: "search a travel" and "book a travel". These two activities are reachable through one URL, "http://flight-travel-service", (see Figure 1.1(a)). In `RESTful`, this service is represented as two resources: "travel" resource and "booking" resource. Each resource has four operations: get, put, post and delete. To search a travel, the client should call the get operation on the "travel" resource while to book a travel, the client should call the put operation on the "booking" resource. Each resource is accessible through a specific

[1] http://www.programmableweb.com/api/flickr
[2] http://www.programmableweb.com/api/ebay
[3] http://www.programmableweb.com/api/paypal

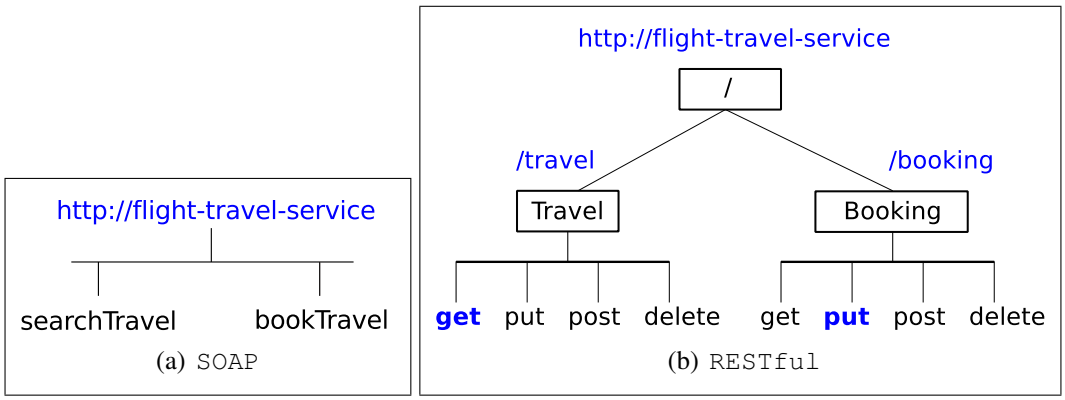


Figure 1.1: Flight reservation service abstraction with SOAP and RESTful

URL which is a continuation of a base URL of the root resource under which are localized the two sub-resources: "travel" and "booking", (see Figure 1.1(b)).

Actually, the use of object-oriented (OO) languages in the implementation of Web services is increasing for two reasons: (i) the object languages are known by most developers, (ii) Web services promote an environment for distributed systems that is loosely coupled and interoperable. Using these frameworks, developers can easily transform an object code into a Web service, or access a remote Web service. For instance, for the flight reservation service, a Java developer considers a Java interface defining an operation, book, which takes in parameter an instance of class Ticket (for simplification reasons, we consider **void** as a return type of book). To deploy his Java code as a Web service, the developer can convert, using the framework, the Java interface to a standardized structural interface: WSDL (for SOAP) or WADL (for RESTful). This structural interface depends on a schema where the structural Ticket type is defined. A client who wishes calling this service, should recover the structural interface to generate, using the framework, a Java interface and the corresponding classes. Figure 1.2 depicts this example.

As different OO programming languages and different service models exist, several frameworks exist like: cxf, RESTEasy, RESTlet, Systinet, .Net, etc. In this thesis, we often refer to the cxf framework because: (i) it is a popular framework under an Apache License, (ii) it allows the development of both service models, RESTful and SOAP, (iii) it uses Java language (iii) it is an implementation of standards (JAX-RS for RESTful and JAX-WS for SOAP).

As shown in Figure 1.2, the existing OO frameworks for Web services implementation present two levels: an object level and a service level. In such a context, different problems exist due to the differences between the object world and the service world. Each level has its own conceptual architecture: the object level belongs to the Distributed Objects Architecture (DOA) [24] while the service level belongs to the Service Oriented Architecture (SOA) [27]. Each architecture has its principles:

- DOA supports the *substitution* principle [51] which allows different substitutions to take place in the object level: a value of a subtype could be exchanged between the client and the server where a value of a supertype is expected. We distinguish two kinds of substitution:

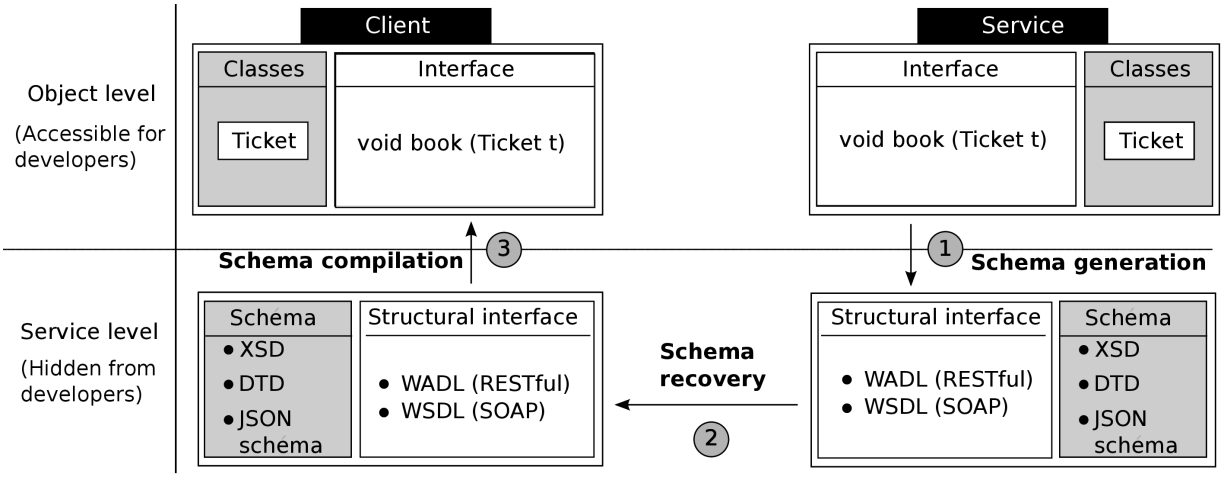


Figure 1.2: Development example: Flight reservation

Value Substitution. Figure 1.3(a) represents the interface of a service composed of one operation (**void** `op(A a)`) and hierarchies of data classes, on the server side and on the client side, before and after a refinement. After the generation of the client proxy from the contract deployed on the server, 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`.

Interface Substitution. Figure 1.3(b) represents two services and a unique client. The client is initially configured to call $Service_1$. $Service_1$ is then replaced with another one, $Service_2$, 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. Applying the substitution principle, the client can switch from $Service_1$ to $Service_2$.

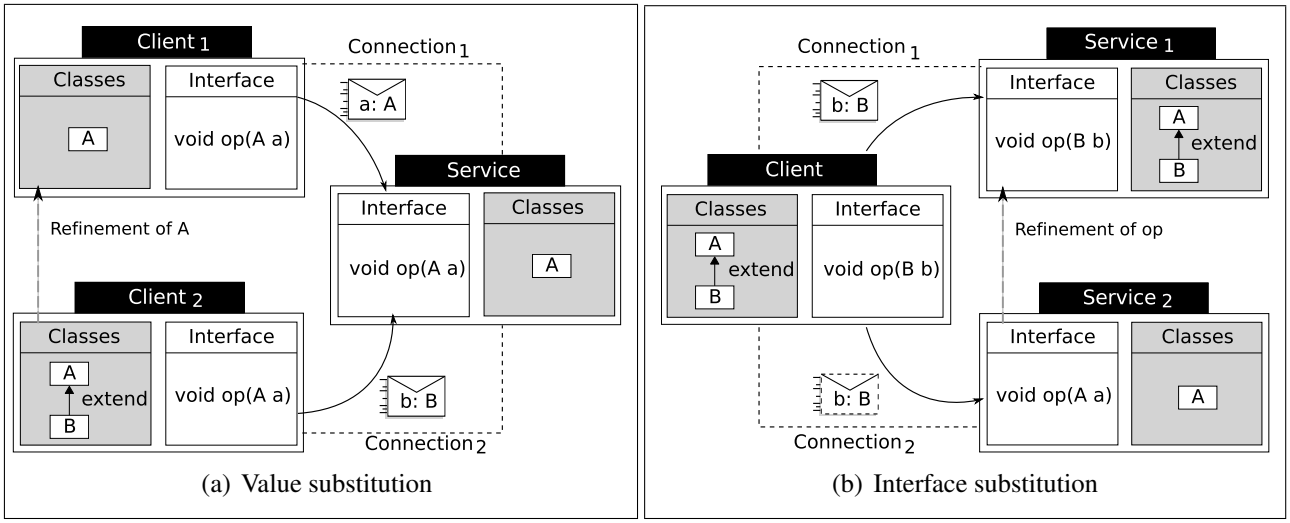


Figure 1.3: Substitution principle by examples

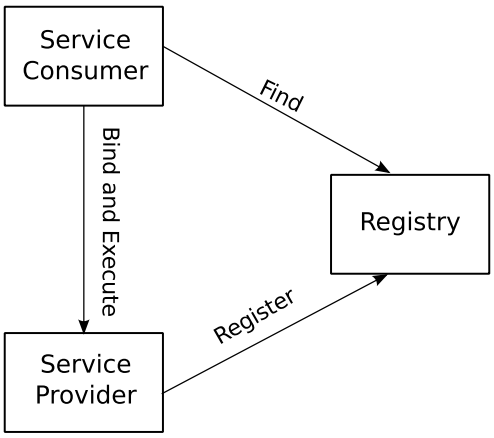


Figure 1.4: The triplet Client/Server/Registry in the SOA architecture

- SOA is based on a triplet: client/server/registry (See Figure 1.4):
  - Service providers publish the availability of their services
  - Service brokers register and categorize published services and provide search services
  - Service requesters use broker services to find a needed service and then use that service

  Different Web service technologies actually exist: SOAP defines UDDI [8] and WS-Discovery [1] standards, while RESTful defines Linked Data [37, 61] standard.

The question which we can rise here is that:

Knowing the architectural differences between the object level and the service level, how these two levels should be connected together while respecting the characteristics of each one?

The existing OO frameworks for Web services were built in an operational way without following a well defined semantic specification. Therefore, once a developer wishes to apply a characteristic specific to SOA or DOA, things seem to be complex and the system ends by breaking down. In the following we present briefly three main problems to be discussed in this thesis and which are due to the mismatch between these two different architectures.

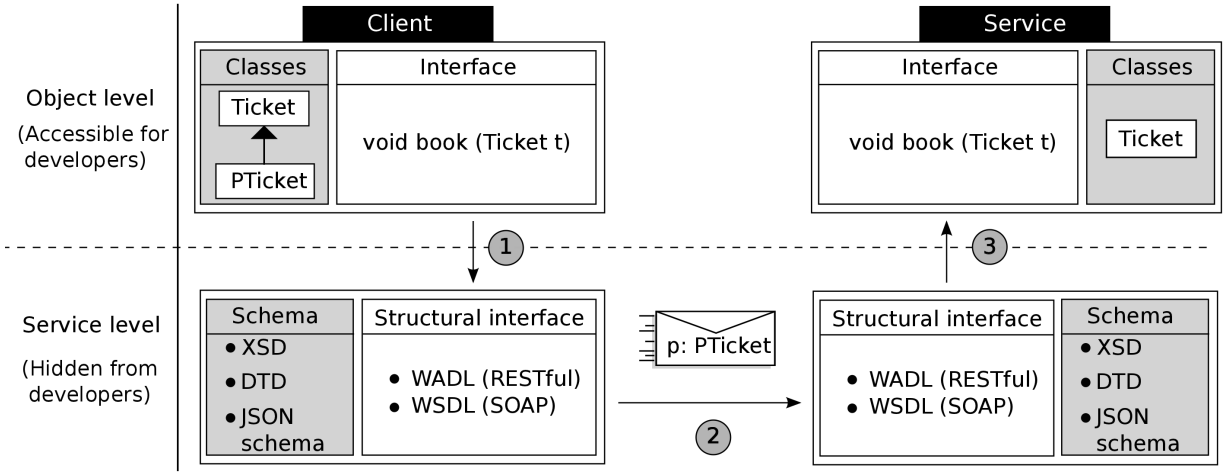


Figure 1.5: Subtyping problem

Subtyping problem. Let us come back to the example of the flight reservation service but this time we consider that the client would like to add some preferences on his request to book a ticket. The service can consider these preferences or not. Let us take the case when the Web service does not treat such particular preferences but treats all requests similarly. In term of OO programming, at the client side, a ticket with preferences could be represented as an instance of a `PTicket` class which extends the `Ticket` class. At the server side, only the `Ticket` class is known. The scenario is depicted in Figure 1.5. This example belongs to the substitution principle as previously described. Testing this example on the famous `cxf` framework gives errors. This problem is due to the *data binding* tools, used by the existing OO frameworks, to convert objects into documents and inversely, we can cite for instance `JAXB` and `Aegis` for `XML`, or `Jettison` and `Jackson` for `JSON`. These tools were not initially dedicated to connect the object level and the service level for the simple reason that they do not consider the substitution principle. Indeed, the substitution principle is not at all considered in the existing OO frameworks for Web services: classes of the development environment are twinned between the client and the server. This similarity ensures a successful interoperability for Web services but requires a tight coupling between clients and servers. Moreover, a big part of the complexity of the data binding tools is due to that: starting from a complex structural type (like `XML` Schema), how to convert it into a corresponding object type. However, the reverse sense is mainly required for the OO frameworks : starting from implemented classes, a service provider deploys its code as a structural service interface, which is then used by the client to generate the corresponding classes. Therefore, a lot of complexity in these tools is useless in such kind of frameworks.

Discovery based on the interface substitution problem. Coming back to the interface substitution scenario defined in Figure 1.3(b), the question is: how the client can know that $Service_2$ is subtype of $Service_1$. Indeed, based on the `SOA` fundamental principle based on the triplet client/server/registry, the client should ask first a registry that is able to de-

tect $Service_2$ as a subtype of $Service_1$. The need of discovery with subtyping has been discussed by several work [50, 47], however, it is completely missing in the existing standards like `UDDI` and `WS-Discovery`.

OO discovery programming problem. Again, we consider the interface substitution scenario defined in Figure 1.3(b), this time we focus on the interaction between the client and the registry in order to discover first $Service_1$ than $Service_2$. The context is represented in Figure 1.6.

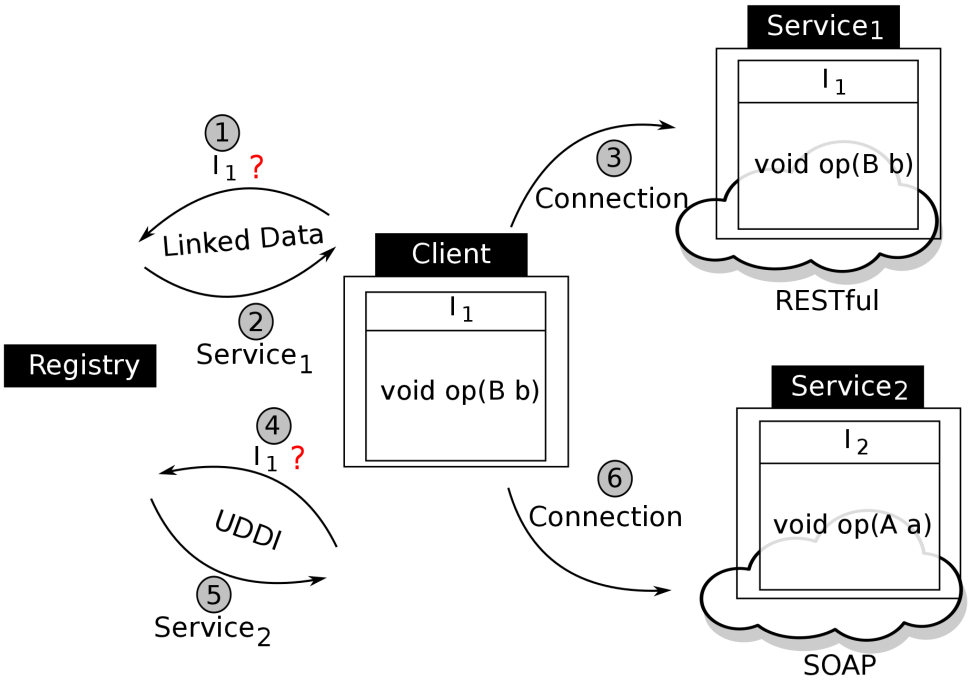


Figure 1.6: Different discovery APIs

First, the client asks the registry for a service providing the required interface I_1. The registry sends a reference to $Service_1$. The client connects to $Service_1$. In a second time, the client asks again the registry for a service providing I_1. The registry sends a reference to $Service_2$. The client makes a new connection to $Service_2$. In this example, we suppose that $Service_1$ is a `RESTful` service and $Service_2$ is a `SOAP` service. Therefore, to discover $Service_1$, the client should use a linked data protocol, while to discover $Service_2$, a `UDDI` protocol for instance is required. These two discovery standards have different degrees of difficulty and this difficulty appears at the object level using the existing defined APIs for these standards. Consequently, discovery development becomes a real complex task specially when the discovery protocol or the model changes, then the object code should suffer from deep modifications. However, despite the differences between these standards, there is one common goal: searching a service using some descriptive parameters and getting then a convenient service location. Therefore, there is a need to unify

concepts at the service level in order to make discovery technical details transparent at the object level.

The big picture of this problematic is clearer if we compare it to a more concrete domain, like mechanic. Imagine that you are interested in building a car. Instead of starting by a well designed plan, you start collecting parts from here and there to make it. In result, you will have a rolling care, but it will be most likely similar to the one presented in Figure 1.7. That is how OO frameworks for Web services actually look like.

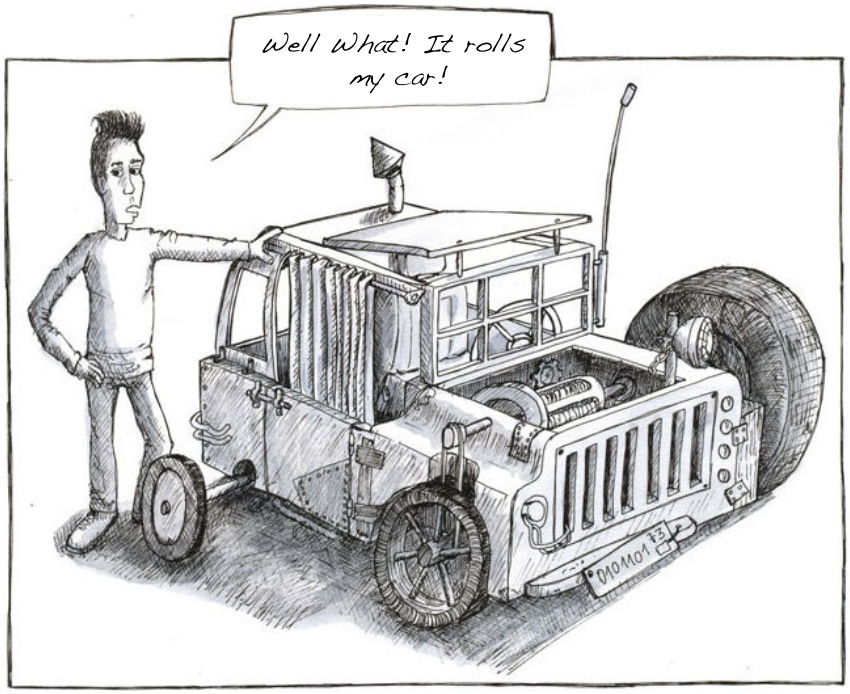


Figure 1.7: How Web services in object-oriented frameworks actually run [4]

In order to avoid problems before they occur when it will be too late, specially that the number of applications based on OO Web services is greatly increasing, we provide in this thesis a specification for such frameworks. We aim to improve these frameworks in order to allow a unification of dynamic discovery with subtyping and to allow an interoperability by respecting OO subtyping between required and provided services. We focus mainly on two properties in the specification of these frameworks:

- First a loose coupling between the two levels, which allows the complex technical details of the service level to be hidden at the object level and the service level to be evolved (from a `RESTful` model to a `SOAP` model and inversely) with a minimal impact on the object level,

[4] This picture is inspired from www.luc-damas.fr: http://www.luc-damas.fr/humeurs/msieur-il-marche-mon-programme/

- Second, an interoperability induced by the substitution principle associated to subtyping in the object level, which allows to freely convert a value of a subtype into a supertype.

This thesis is composed from five parts:

Part I - State of the Art. In this part, we present the state of the art and discussions around it in three chapters:

- In Chapter 2, we present the SOA principle entities and characteristics based on the triplet: client/server/registry. We then present how SOAP and RESTful fit with the SOA principles using the existing standards. We discuss in this chapter how much the existing Web services implementation practices respect dynamic discovery in SOA.
- In Chapter 3, we provide pertinent abstractions of the data flow and the control flow in an OO framework to explain how the framework processes data from the top object level to the network level and conversely. Then, in order to compare the OO frameworks of Web services with the most prominent examples of OO middleware (like CORBA and RMI), we introduce the main concepts of distributed objects engineering. We discuss in this chapter the possibility to evolve the existing OO frameworks for Web services by considering two criterions: The distributed objects engineering and the SOA architecture.
- In Chapter 4, we focus on the state of the art of modeling a unified *black box* model, where a service is represented as a black agent implementation with a structural interface. Such a model is useful to conceptually unify the service level in order to offer development facilities at the object level. We are interested in work around modeling dynamic discovery which makes the network topology evolve. We present also some work comparing between the two Web services models, SOAP and RESTful. Moreover, in order to unify the type of exchanged messages in SOA which supports contents for service discovery, we discuss a well known formal type system. This type system supports dynamic discovery with type inference and subtyping. We compare it to the existing used type systems for Web services to show their weaknesses. We discuss in this chapter the existing work for type safety and type checking.

Part II - Towards a Well-Founded Object-Oriented Framework for Web Services. In this part, we go more in details to explain by examples the problems in the existing OO frameworks for Web services. Before going to present our proposed solution to fix these issues, we introduce a black-box model which unifies the concepts in Web services models and which defines a safe type system with subtyping. This part is composed from two chapters:

- In Chapter 5, we present more in details the requirements in OO frameworks: Loose coupling and substitution principle. We show by examples applied on some existing frameworks how these requirements are not fully satisfied.

- In Chapter 6, we provide a formalization of a black-box model for service communications inspired from the state of the art as presented in Chapter 4. This model supports message-oriented services in the presence of discovery and subtyping. Moreover, we present a type system which includes communication using typed first-class channels, general set operators over types and a subtyping relation. We show also the soundness of our type system, even in the presence of attackers and insecure channels.

Part III - A New Specification for a Well-Founded OO Framework for Web Services. In this part, we present our specification to resolve the existing problems in Web services frameworks in two chapters:

- In Chapter 7, we present how the existing service discovery standards could be unified for `SOAP` and `RESTful` using our unified model presented in Chapter 6. We show how the concepts of the unified formal model could fit with the OO world and how an OO API for dynamic discovery should be built conformally to the OO development practices.
- In Chapter 8, we provide a new specification of the data binding used to convert between objects and documents in order to resolve problems presented in Chapter 5. We show how this specification can be concretely implemented in `cxf` using `JAXB` data binding. Moreover, we show how some complex configuration details required by the data binding could be hidden at the object level by considering `JAXB`. We finish this chapter by discussing the advantages of our solution and its performance.

Part IV - Perspectives. We conclude this thesis in Chapter 9 by summarizing our contributions. Then we discuss limitations of this thesis and future work in Chapter 10.

Part V - Appendix. We present in this part some clarifying details about the existing frameworks and our proposed solution.

Publications and Research work

Conference papers

1. Diana Allam, Hervé Grall, and Jean-Claude Royer. **The Substitution Principle in an Object-Oriented Framework for Web Services: From Failure to Success**. iiWAS 2013 -The 15th International Conference on Information Integration and Web-based Applications & Services, Vienna, Austria, December 2013. ACM, ISBN 978-1-4503-2113-6.

2. Diana Allam, Hervé Grall, and Jean-Claude Royer. **From Object-Oriented Programming to Service-Oriented Computing: How to Improve Interoperability by Preserving Subtyping**. In Karl-Heinz Krempels and José Cordeiro, editors, WEBIST 2013 - 9th

International Conference on Web Information Systems and Technologies, Aachen, Germany, March 2013. SciTePress Digital Library.

3. Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, and Mario Südholt. **A Message-Passing Model for Service Oriented Computing**. In Karl-Heinz Krempels and José Cordeiro, editors, WEBIST 2012, 8th International Conference on Web Information Systems and Technologies, Porto, Portugal, pages 136-142, April 2012. SciTePress Digital Library.

Research reports and deliverables

1. Diana Allam et al. **Mechanisms for property preservation**. Deliverable D2.4, CESSA ANR project, no. 09-SEGI-002-01, Jully 2012.

2. Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, and Mario Südholt. **Well-Typed Services Cannot Go Wrong**. Research Report RR-7899, INRIA, May 2012.

3. Diana Allam et al. **Extension of the service model for security and aspects**. Deliverable D1.3, CESSA ANR project, no. 09-SEGI-002-01, September 2011.

4. Diana Allam et al. **Model and formal architecture specification**. Deliverable D1.2, CESSA ANR project, no. 09-SEGI-002-01, January 2011.

Posters and oral presentations

1. Diana Allam. **A Unified Formal Model for Service Oriented Architecture to Enforce Security Contracts**. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development, editor, AOSD 2012 Student Research Competition (Poster), Potsdam, Germany, pages 9-10, March 2012. ACM.

2. Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, and Mario Südholt. **The Synthesis Problem for Trusted Service-based Collaborations**. In Actes des troisièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel, Lille, France, June 2011.

3. Diana Allam, Hervé Grall, and Jean-Claude Royer. **Towards a Unified Formal Model for Service Orchestration and Choreography**. In Olivier Caron Yves Ledru, Anne-Françoise Le Meur, editor, Actes des troisièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (Poster), Lille, France, June 2011.

State of the Art

2

Service Oriented Architecture and Web Services

Contents

Web services promote an environment for systems that is loosely coupled and interoperable. Many of the concepts for Web services come from a conceptual architecture called Service-Oriented Architecture (SOA). This chapter describes in Section 2.1, the SOA principle entities and characteristics based on the triplet: client/server/registry. We show that the dynamic Web services discovery is an important property in such an architecture in order to maximize loose coupling and reuse. Then in Section 2.2, we present the two Web services implementation models, SOAP and RESTful, by showing how they fit with the SOA principles using the existing standards. Finally, we discuss in Section 2.3 how much the existing Web services implementation practices respect the core characteristic of SOA: dynamic discovery.

2.1 Service Oriented Architecture (SOA) concepts

SOA [27] is a lightweight environment for dynamically discovering and using services on a network. It is based on the principle of separating the service implementation from its interface. Services are seen simply as endpoints that treat consumers requests following a specific contract represented by its interface. The service implementation technology and the way it executes tasks is completely transparent to the consumers. The SOA characteristics are mainly based on two principles [40, 54]:

- **Interoperability:** It is the ability of systems using different platforms and languages to communicate with each other. Each service provides an interface that can be invoked through a connector type. An interoperable connector consists of a protocol and a data format that each of the potential clients of the service understands. Interoperability is achieved by supporting the protocol and data formats of the service clients. Techniques for supporting standard protocol and data formats consist of mapping each platform characteristics and language to a mediating specification. The mediating specification maps the formats of the interoperable data format to the platform-specific data formats.

- **Loose coupling**: SOA promotes loose coupling between service consumers and service providers. A system degree of coupling directly affects its modifiability. The more tightly coupled a system is, the more a change in a service will require changes in service consumers. Coupling is increased when service consumers require a large amount of information about the service provider to use the service. If the consumer of the service does not need detailed knowledge of the service before invoking it, the consumer and the provider are more loosely coupled. A consumer asks a third-party registry for reference about the service it wishes to use.

In the following, we discuss some properties of SOA induced by the interoperability and loose coupling principles.

2.1.1 Interface-based interaction

The dependencies and communications between consumers and the service should be limited to consumers conformance to the service contract (interface) [54]. The separation between the service interface and its implementation allows services to interact without needing a common shared execution environment. Due to this opacity, services become more autonomous since they are able to freely choose models and languages, implementation environments and to substitute one service implementation for another.

The service interface should describe the aspect of a service that allows a potential service consumer to understand and evaluate its capabilities. This interface defines the service functions. Moreover, related constraints and policies on the service provided operations could be also represented in the interface. All these aspects contribute to the definition of the service formal contract, which is shared between the users and the service itself.

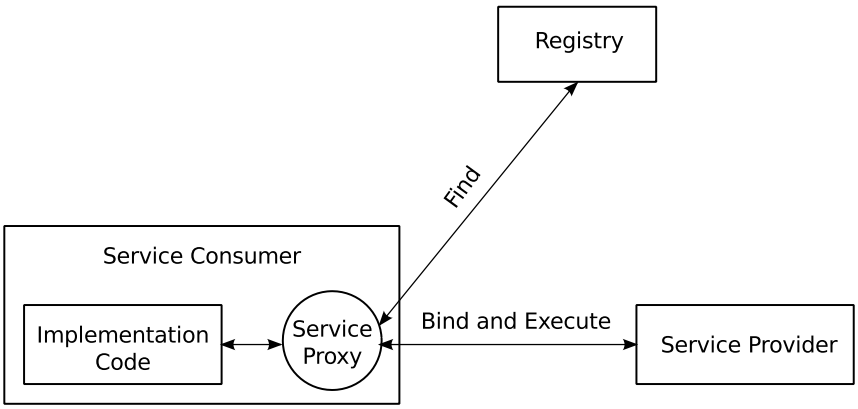


Figure 2.1: Service proxy [54]

2.1.2 Dynamic discovery

SOA supports the concept of service discovery. At runtime, a service consumer asks a registry for a service. The best way to explain dynamic discovery is to use an example. A flight reservation application (consumer) asks a registry for a service that performs flight booking. The registry returns all entries that support such a service. The entries also contain information about the service, including booking fees. The consumer selects the service (provider) from the list based on the lowest booking fee. Using the service reference from the registry entry, the consumer binds to the provider of the flight reservation service using a proxy (see Figure 2.1). The service consumer executes the request by calling an API function on the proxy. The proxy is simply a local reference to a remote service implementation. If the proxy changes the interface of the remote service, then technically, it is no longer a proxy. A service proxy is written in the native language of the service consumer. For instance, a proxy for a well defined service provider may be in Java or Visual Basic.

A Web service is basically referenced with two aspects: its interface and its access information (URI, transport protocol, etc.) [54]. Switching dynamically to a new Web service is possible by changing one or/and the other aspect. Therefore, we can deduce that two service discovery methodologies exist:

- **Updating service access methodology:** it requires modifying the service access information each time a switch is done while keeping the required interface unchanged. This discovery search is based on sending some information about the required interface (a kind of interface representation) which guides the registry to identify it and therefore, to refer the corresponding service providers. Following this method, the service consumer proxy has only to update the reference access to the remote service each time a switch is done. The required interface to which the proxy is linked remains unchanged.

- **Interface generation methodology:** It needs the generation of the required interface each time a switch is done. The dependency is a runtime dependency and not a compile-time dependency. All the information the consumer needs about the service is obtained and used at runtime. Clients do not need any compile-time information about the service. The

service interfaces are discovered dynamically. The service consumer does not know the format of the request message or response message or the location of the service until the service becomes needed.

The existing state of the art work around dynamic Web services discovery for SOA is mainly focused on the second methodology, "Interface generation methodology". The discovery search is most likely based on semantic equivalence between services, like in [85, 78, 84, 46, 69]. Following this method, the service consumer proxy is regenerated each time a switch is done to a new Web service. For instance, Zisman et al. in [85] define a framework for dynamic service discovery based on a specific metric to compute the distance between a query and a candidate service. This metric is based on a service representation following its structure, behavior, quality, and contextual characteristics of a system represented in query languages.

On the other side, state of the art work around the "Updating service access methodology" remains theoretical. Castagna et al. work [29] for instance defines a type system dealing with functions. This type system could be used by considering functions as references for Web services operations, known as `channel` in the classic process calculi, π-calculus [70]. Castagna et al. in [19] present a variation of their type system applied to extend asynchronous π-calculus [16] with semantic characterization of channel types and a semantic subtyping. In such formal models, service locations are represented as channels in a message content. The type of this channel will be deduced dynamically at reception: the type of the discovered channel is configured at compile time. This mechanism is known by `type inference` [75]. Moreover, the "Updating service access methodology" has also sense by considering subtyping: a client can switch dynamically from a service to a more specific one without requiring any modifications on its implemented code. This subtyping requirement has been discussed by Kourtesis et al. in [47]. The type system of Castagna et al. [29], supporting also service subtyping becomes a theoretical foundation to improve the "Updating service access methodology". Castagna et al. work will be discussed more in details in Chapter 4.

2.1.3 SOA entities

The two previous properties, using contracts and dynamic discovery, are abstracted in Figure 2.2 using the triplet, service provider, service broker and service requester [54]:

- Service providers publish the availability of their services
- Service brokers register and categorize published services and provide search services
- Service requesters use broker services to find a needed service and then use that service

The service contract (or interface) is a specification of the way a consumer of a service interacts with the provider of the service. A service contract may require a set of preconditions and postconditions. The contract may also specify Quality of Service (QoS). QoS levels are specifications for the nonfunctional aspects of the service. For instance, a quality of service attribute is the amount of time it takes to execute a service method.

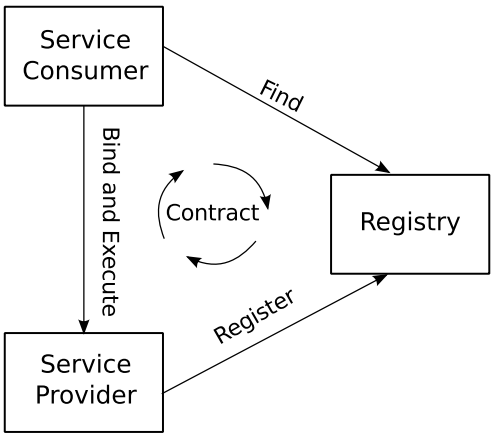


Figure 2.2: The triplet Client/Server/Registry in the SOA architecture

2.2 Web services

Web services are simply one set of technologies that can be used to implement SOA successfully [55, 38]. Systems that have to communicate with other systems use communication protocols and the data formats that both systems understand. Today, service-based applications can be built according to two competing models: SOAP and RESTful. Both have similar characteristics but architectural decisions and targeted applications are different [64]. Today, SOAP and RESTful models are often supported at the same time. Figure 2.3 presents a statistic on the used protocols by existing APIs to call provided Web services. For instance, Google Translate provides a RESTful API[1] that can be used by Web applications to include the translation Web service. Google itself provides a publicly available Web application (at http://translate.google.com) that uses the Web service. Similarly, PayPal provides an API (for both RESTful and SOAP protocols)[2] which offers online payment solutions. Figure 2.3 clearly shows that RESTful invades the market (70% of the existing service APIs use RESTful protocol). In the following, we present the SOAP and the RESTful models conformally to the following characteristics of SOA: Interface description, transport protocol and discovery.

2.2.1 SOAP model

SOAP, also known as process-oriented Web services, supports strong contracts, *e.g.,* for security and transactional properties, and disposes of a larger tool base for development, execution and maintenance. SOAP had led to the development of the technology of big Web services by defining services interface via a Web services Description Language (WSDL) [4] accessed through a standard protocol like the Simple Object Access Protocol (SOAP).

[1] https://developers.google.com/translate/
[2] http://www.programmableweb.com/api/paypal
[3] http://www.programmableweb.com/apis
[4] http://www.w3.org/TR/wsdl

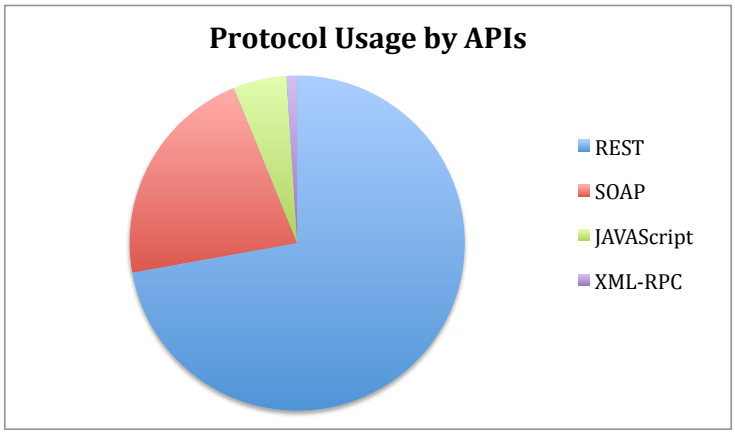


Figure 2.3: API Protocols[3]

Interface description. WSDL is an XML-based standard specification for describing SOAP Web services. The structure of a WSDL (1.1 version) document is presented in Figure 2.4. The service is represented by a port which defines the address (URI[7]) of the service. The communication protocol and the exchanged data format for the port are represented in a Binding. Multiple ports can be associated to a service if this one is accessible through different addresses or different bindings. The binding information is appropriated to the service operations defined in the interface (portType). This interface defines a set of operations. Each operation has an input message and an output message. Each message is divided into parts such that each part has a type defined in an XSD schema.

Web services definitions can be mapped to any implementation language, platform, object model, or messaging system. As shown in Figure 2.5, WSDL documents consist of two main parts:

- The service interface definition describing the abstract type interface and its protocol binding, known as the WSDL binding document,
- The service implementation definition describing the service access location information, known as the WSDL service document.

Often, WSDL documents are represented as a single file. This file contains both the service interface and implementation documents.

Transport protocol. When a service consumer invokes a Web service, the method invocation is represented as a SOAP[8] message. This message is transmitted through a transport such as

[5] http://www.herongyang.com/WSDL/WSDL-11-Introduction-What-Is-WSDL-11.html

[6] http://help.eclipse.org/juno/index.jsp\?topic=%2Forg.eclipse.jst.ws.consumption.ui.doc.user%2Fconcepts%2Fcwsdlud.html

[7] As pointed out by Berners-Lee [10], the URI, a compact string of characters used to identify or name a resource, is the most fundamental specification of Web architecture. The URI specification transforms the Web into a uniform space, making the properties of naming and addressing schemes independent from the language using them.

[8] http://www.w3.org/TR/soap/

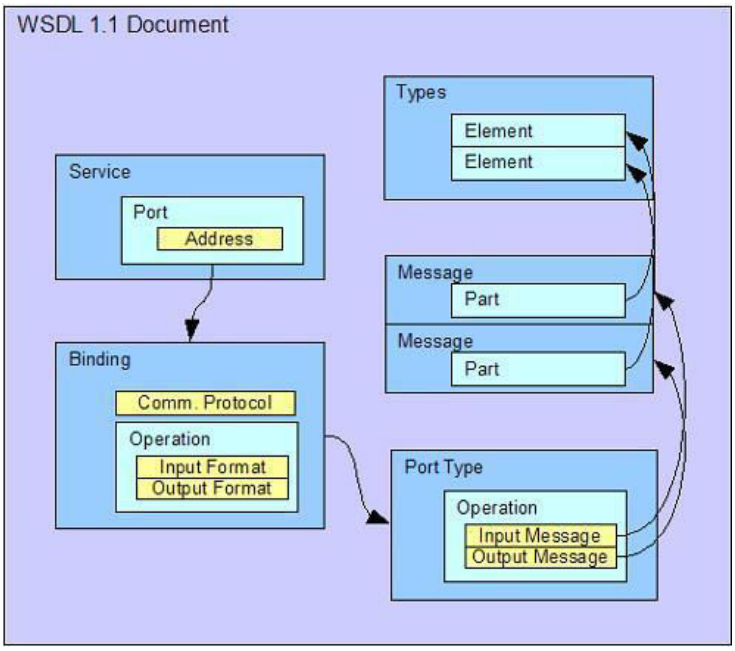


Figure 2.4: WSDL structure[5]

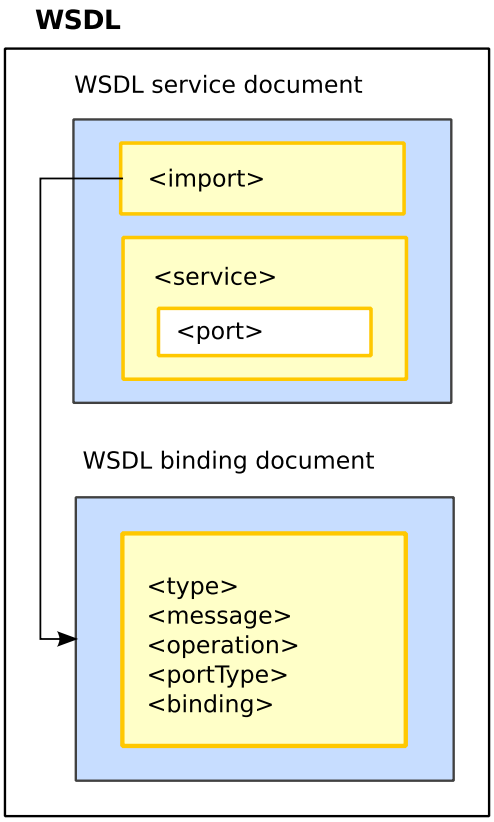


Figure 2.5: WSDL abstract and concrete parts [6]

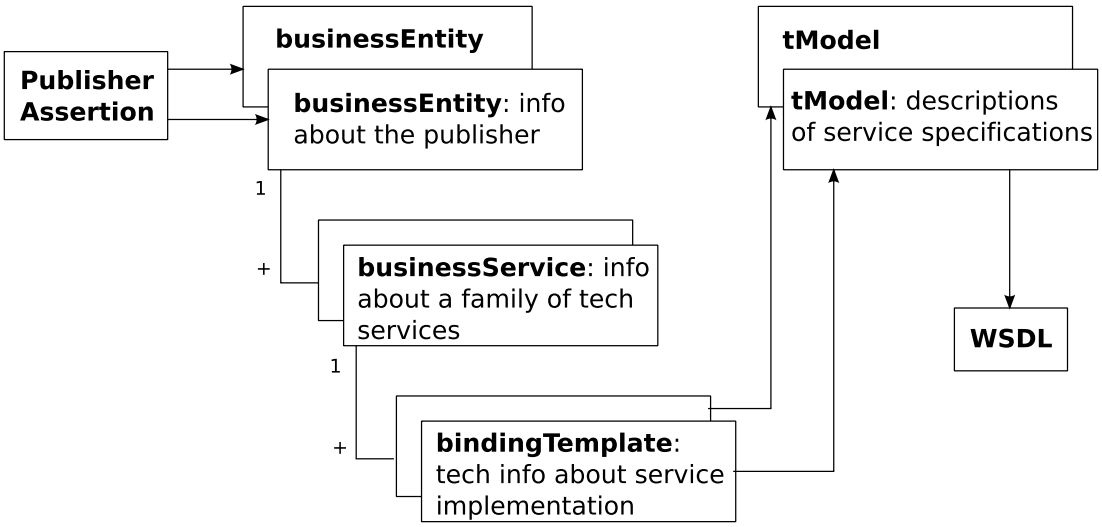


Figure 2.6: UDDI entities [8]

http or SMTP, to the service provider, which parses the message into a method invocation. After the service provider executes the client request, the reply is parsed into a SOAP response message to be transmitted through a transport to the client. The SOAP specification was designed to unify proprietary Remote procedure Call (RPC) communication, basically by serializing into XML, the parameter data transmitted between components, transporting, and finally deserializing them back at the destination component. SOAP is fundamentally a stateless, one-way message exchange paradigm. At its core, a SOAP message has a very simple structure: an XML element with two children elements, one containing the header and the other the body. The header contents and body elements are also represented in XML. The header is an optional element that allows the sender to transmit control information. Headers may be inspected, inserted, deleted, or forwarded by SOAP nodes encountered along a SOAP message path.

Discovery. Two discovery standards are defined for SOAP service model:

- UDDI standard: the Universal Description, Discovery, and Integration (UDDI) [8] standard can be used if a centralized registry is appropriate to discover the access points of services that are known to implement a WSDL interface. UDDI registries are themselves Web services that expose an API as a set of well-defined SOAP messages. UDDI supports two types of conversations:

  - the service provider uses the UDDI directory to publish information about the Web services it supports. Registering a service involves four core data structure types:

    * The businessEntity data type contains information about the business that has a published service.
    * The businessService data type is a description of a Web service.

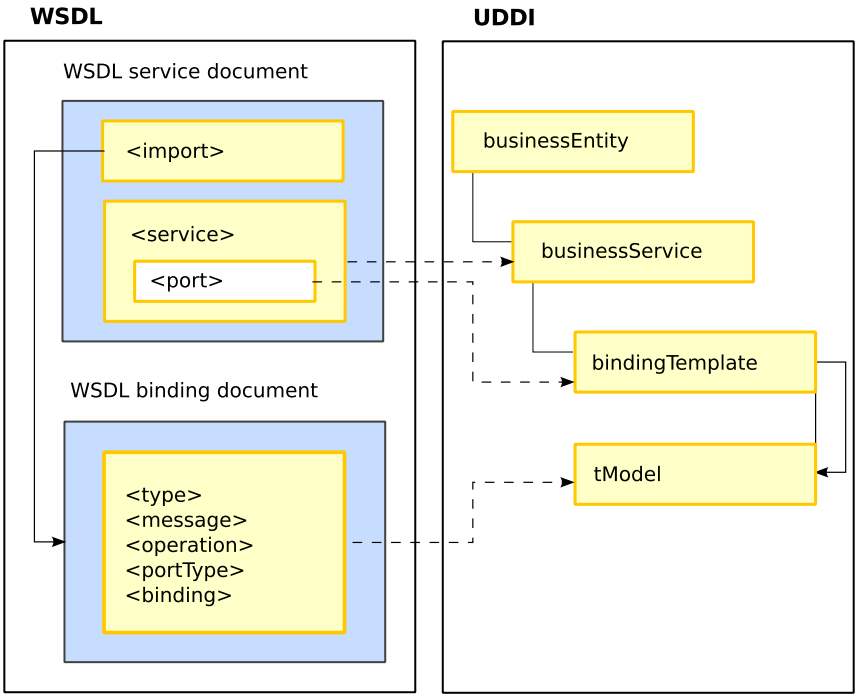


Figure 2.7: Relationship between UDDI and WSDL[9]

* The bindingTemplate data type contains technical information for determining the entry point and construction specifications for invoking a Web service.
* The tModel data type provides a reference system to assist in the discovery of Web services and acts as a technical specification for a Web service.

Figure 2.6 shows the link between these data structure types. A businessEntity is associated to multiple businessService as a publisher can provide multiple Web services. Each businessService is associated to multiple bindingTemplate as a service can be available at different access points. Finally, the Publisher Assertion entity is used to specify relationships among multiple businesses described by different businessEntity entities. For example, a corporate enterprise may have multiple related subsidiaries and each of them may have registered as a businessEntity in the UDDI.

– The Web services consumer sends SOAP-formatted XML messages over http to the UDDI directory, to retrieve a listing of Web services that match its criteria.

  Discovering the access point of a Web service having a specific WSDL binding document (the service interface definition) is possible by getting a bindingTemplate of the tModel associated to the WSDL binding document. This bindingTemplate refers to a location of a service implementation of the

[9] http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jst.ws.consumption.ui.doc.user%2Fconcepts%2Fcwsdlud.html

WSDL binding document. Figure 2.7 illustrates the relationship between UDDI and WSDL. The WSDL service element references the WSDL binding element. The URL of the document containing the WSDL binding element is published to the UDDI business registry as a tModel. The URL of the document containing the WSDL service element is published to the UDDI business registry as a businessService and contains information about the bindingTemplate.

- WS-Discovery: the WS-Discovery [1] standard is a protocol to enable dynamic discovery of services available on the local network. WS-Discovery can be used as part of an ad-hoc mode or a managed mode. In an ad-hoc mode clients and services should send a multicast Hello message when they join the network and a multicast Bye message when they leave the network. To discover a service, a multicast discovery message is sent. Response messages are sent unicast. A discovery proxy (or a registry) could be placed to facilitate discovery of target services by clients. In a managed mode, Hello and Bye messages are sent unicast between a service provider and a discovery proxy. Clients also send unicast discovery messages to the discovery proxy. This method can reduce the traffic in an ad hoc network. In order to abstract the WS-Discovery mechanism conformally to the SOA architecture based on the triplet (service consumer/service provider/registry), we present in Figure 2.8 the principle discovery steps in a managed mode:

  1. a service provider sends a Hello message to a registry when it joins the network,
  2. a client sends a Probe request to the registry to locate services,
  3. the registry replies with a ProbeMatch response containing matching target services if any,
  4. the service provider sends a Bye message to the registry when it leaves the network.

  A probe includes zero, one, or two constraints on matching target services: a set of Types (*e.g.*,, a WSDL 1.1 portType) and/or a set of Scopes (an extensibility point that allows Target Services to be organized into logical groups). A type T1 in a probe matches with a type T2 of a target service if the QNames match. The QName represents a qualified name as defined in the XML specifications.

  Specifically, T1 matches T2 if all of the following are true:

  - The namespace [Namespaces in XML 1.1] of T1 and T2 are the same.
  - The local name of T1 and T2 are the same.

  The reply ProbeMatch message contains zero or multiple EndPoint Reference to refer to services satisfying the client request. An EndPoint Reference is a standardization specified in the WS-Addressing specification[10] to represent a service which contains the location of a specific service.

[10] http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/

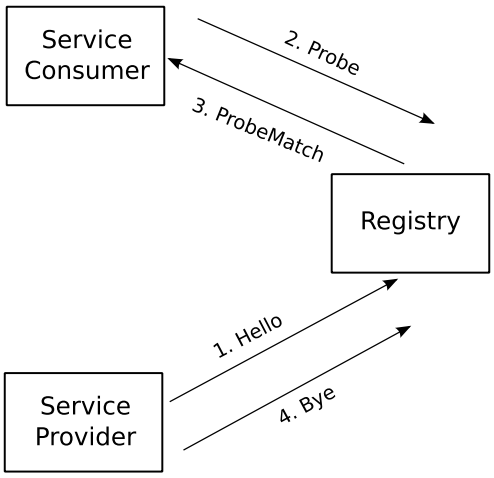


Figure 2.8: WS-Discovery using a registry

2.2.2 RESTful model

RESTful [67], also known as resource-oriented Web services, enables the more lightweight implementation of service-based applications in terms of traditional languages and infrastructures. It returns to the original design principles of the World Wide Web and its RESTful style by defining a CRUD (Create/Remove/Update/Delete) interface accessed directly via http protocol.

Interface description. RESTful does not have a standard structural service interface like WSDL for SOAP. Sometimes, description interfaces are defined for human use, but RESTful services start using a Web Application Description Language (WADL) [34] interface which could be similar to WSDL for SOAP. WADL is an alternative service description language that is more in line with the Web.

The structure of a WADL document is represented in Figure 2.9. This figure shows only the basic structure of a WADL document. Other details are hidden here and could be found in the WADL specification [34]. The RESTful interface should be a hierarchy of resources. The WADL describes how one resource links to another. Each resource is reachable through a unique address. Four operations can be used to manipulate resources:

- PUT: Used to add or update a resource. It is idempotent, so if PUT is applied on a resource twice, it has no effect.
- POST: Used to modify and update a resource. It allows requests to update different parts of a resource at the same time.
- GET: Used to get a resource
- DELETE: Used to delete a resource.

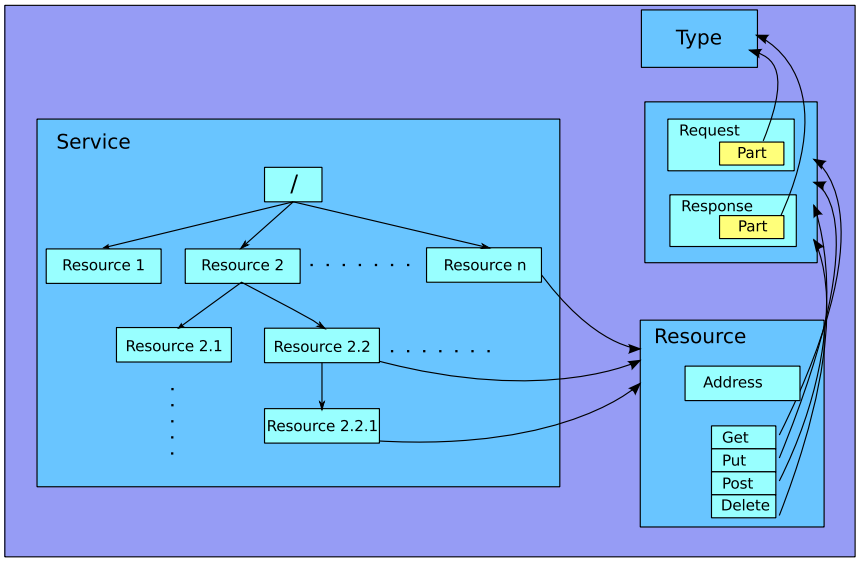


Figure 2.9: WADL structure

Each operation defines a request and a response part. The request describes the input to the method as a set of parts[11]. In the same manner as the request part, the response describes the possible outputs of the method[12]. Types of request and response elements are defined in an XSD schema or in a JSON schema. Complex data structures could be represented in JSON without the need of a schema.

Transport protocol. As we have explained before, exchanged data in SOAP services are packaged in SOAP messages. These messages are almost always sent over http. Most SOAP services support multiple operations on diverse data, all mediated through POST on a single URI. This is not resource-oriented: it is RPC-style. In RESTful services, exchanged data are packaged in a payload of an http request method. A service is split into resources: every "thing" in the service with a separate URI. On each URI, the client can use the http methods, (GET, PUT, POST and DELETE) as defined previously.

Discovery. Discovery in RESTful services is more likely represented by the search engine, like Google. This helps (human) clients to find the resources they are looking for. Actually, RESTful services could be also discovered dynamically using an intelligent machine processing thanks to Linked Data [37, 61]. Indeed, resources are progressively linked together to alleviate the information overload and to increase the information accessibility. In order to establish such links, resources are documented: there is a vocabulary of relations to link them with one another. Resource links defined inside resources themselves indicate where sub-resources, related resources and operations are located. Multiple formats already exist for Linked Data,

[11] See section 2.9 of the WADL specification http://www.w3.org/Submission/wadl/
[12] See section 2.10 of the WADL specification http://www.w3.org/Submission/wadl/

such as JSON-LD [77] and Atom links as defined for instance in the RESTEasy framework [2]. The discovery mechanism in RESTful, taking into account such linked resources, requires inserting links in the http response headers or in the message payload. Thus in order to discover a particular resource, a client must at least know a resource linked, directly or indirectly, to the wanted resource.

2.3 Discussion

While Web services provide support for many of the concepts of SOA, they do not respect the triplet: Service consumer/Service provider/Registry. Service consumers can execute Web services directly if they know the service's address and contract. They do not have to go to the registry to obtain this information. Today, in fact, most organizations implement Web services without a registry. This is mainly due to the complexity of the existing discovery standards, like UDDI and WS-Discovery.

Moreover, the existing discovery mechanisms for RESTful and SOAP lack service subtyping as it has been recognized recently in several different contexts [50, 47]. From a theoretical point of view, the equality between declared operations at the client and server is not essential to ensure a correct interoperability. The provided operation by a Web service could be a subtype of the required one by the client, [72]. Hence, according to the variance property 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. The existing discovery standards does not respect this subtyping principle. For instance, when a client needs to switch to a new Web service having a similar role than the old one by keeping the old service interface following the "Updating service access methodology", the client and the new discovered service should have the same interface. In the UDDI standard for example, the discovery search is restricted to services providing the same tModelKey, thus the same WSDL interface. Similarly, in WS-Discovery standard, the two services should have the same QName. Thanks to this restriction, a request at the client side is still valid after switching to the new service. That is due to the fact that, initially, this binding was created according to the interface for the provided service and tightly depends on it.

Therefore, there is a need to include the dynamic discovery, based on subtyping between services, to the existing development practices of Web services. That requires facilitating the use of standards by hiding their complex technical details.

In this thesis, we aim to improve the Web services implementation practices in order to enable dynamic discovery with subtyping, following the "Updating service access methodology". The principle is resumed in Figure 2.10. At compile time (see Figure 2.10(a)), two service providers (Service Provider 1 and Service Provider 2) register their locations at the Registry as providers of a Web service, let us call it S. These two service providers should implement an interface, let us call it I, or a subtype of I, associated to S. The interface I is either specified by the Registry and therefore each service provider has to get this interface to implement it or a subtype of it, or specified by the first service provider which registers an im-

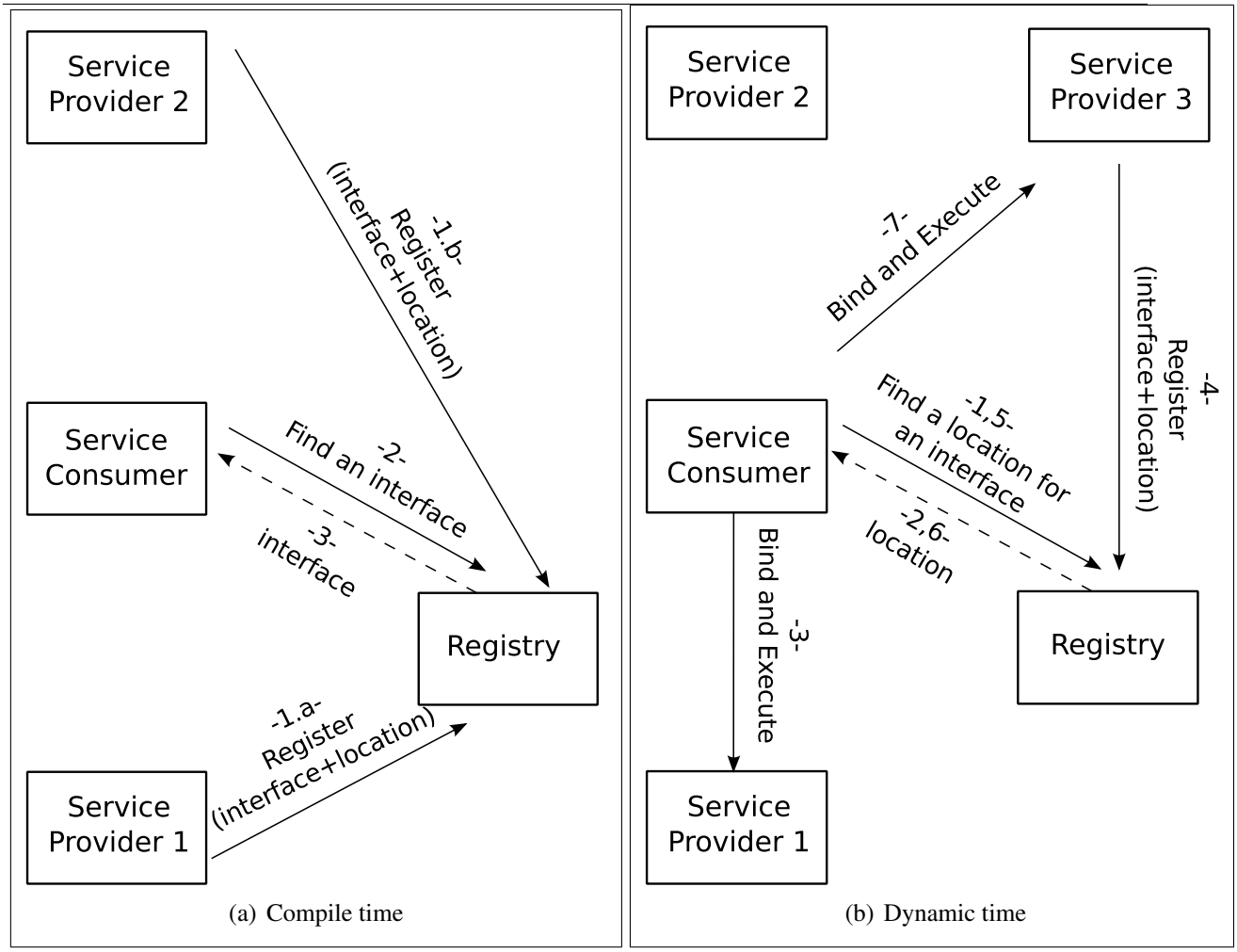


Figure 2.10: Dynamic discovery following "Updating service access methodology"

plementation of the service `S` at the registry, therefore the following service providers have to use this interface or subtype it. Then, the client can ask the registry for a standard service interface of `S`. The client uses this interface to implement its request code to an implementation of `S`, discovered at runtime. The registry replies with the `I` interface without necessarily defining a service location. At dynamic time (execution time), the client asks the registry for a location of an implementation of `I`. The registry returns an available service location from the set of the registered services following some QoS criterias computed at runtime, (in Figure 2.10(b)). The client binds to the discovered service location and executes it. Each time a client wishes to call an implementation of a service `S`, it asks the Registry for an updated address, as new implementations of `S` could be registered also at runtime (see `Service Provider 3` in Figure 2.10(b)). We note that in this thesis, we abstract all aspects of semantical and qualitative analysis between services. We restrict our study to the type of the service interfaces. We consider that the client is connected to services conformally to the scenario of Figure 2.10.

3

Distributed Objects and Web Services

Contents

SOA is built in a very complex manner due to the multiple existing service models, technologies and protocols [5]. This diversity in models and technologies appears at different layers of Web services processing which makes the Web services implementation environment really heterogeneous (see Figure 3.1). Actually, Web services can be implemented in different languages like Java and BPEL (Business Process Execution Language) [60]. Different frameworks already exist in the market for Web services implementation using these languages, like Apache

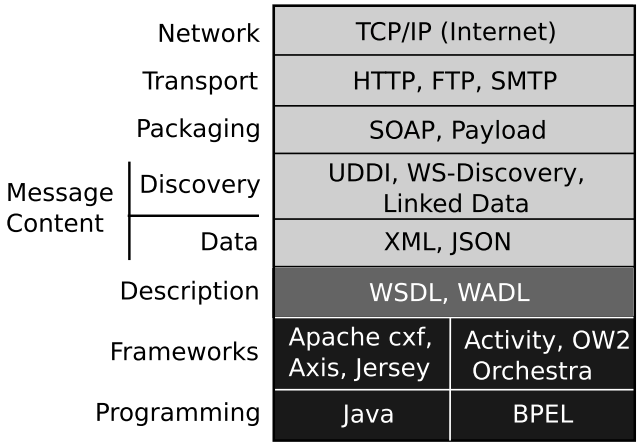


Figure 3.1: An heterogeneous environment

`cxf`[1], `Axis`[2], `Jersey`[3] for `Java` and `Activiti`[4], `OW2 Orchestra`[5] for `BPEL`. Then they could be exposed in a Web services environment by an interface description language (`WSDL` for `SOAP` and `WADL` for `RESTful`) to be reached via a specific protocol with a defined message description (packaging, content, data format, etc.). This heterogeneity does not appear only at providing a Web service, but also at calling it. To call a Web service, different APIs already exist, even the same targeted service could be reached in different ways as it could also be exposed in multiple ways. In this chapter, we focus on the implementation of Web services in `Java` at the server and the client sides, using an OO framework. We choose the well known `cxf` framework as an example which is an implementation of some paradigmatic standards for Web services in `Java`: `JAX-RS API` [65] for `RESTful` and `JAX-WS API` [21] for `SOAP`.

First in Section 3.1, we provide a pertinent abstraction of the data flow and of the control flow in an OO framework to explain how the framework processes data from the top object level to the network level and conversely. Then in Section 3.2, in order to compare the OO frameworks of Web services with the most prominent examples of OO middleware (like `CORBA` and `RMI`), we introduce the main concepts of distributed objects engineering. Finally, we discuss in Section 3.3, the possibility to evolve the existing OO frameworks for Web services by considering two criterions: The distributed objects engineering and the `SOA` architecture.

[1] http://cxf.apache.org/
[2] http://axis.apache.org/axis/
[3] https://jersey.java.net/
[4] http://activiti.org/
[5] http://orchestra.ow2.org/xwiki/bin/view/Main/WebHome

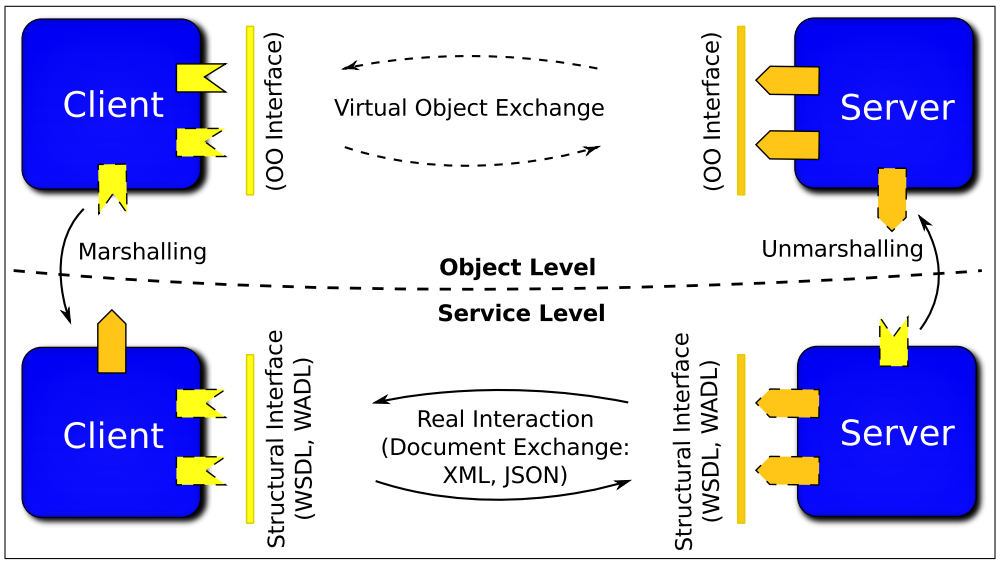


Figure 3.2: Two component levels for Web services communication

3.1 The architecture of an object-oriented framework for Web services

In this section, we present the architecture at two levels, the object level and the service level, in an OO Web services framework. We zoom on the exchanges between the two levels in order to give an abstract view of the framework. We focus on a main component in such frameworks which is responsible of converting objects into structural documents and conversely, called `data binder`. Then, we show how the development and the execution are done in such frameworks. Next, we describe the data flow of messages exchanged between a source and a destination in these frameworks. Finally, we present a control flow (applied to the `cxf` framework) to explain how a framework processes data and how it drives the data binding.

3.1.1 The object level and the service level

In OO frameworks for Web services, 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, an OO framework for Web services like `cxf` contains two levels, one dedicated to objects, the other to services. Figure 3.2 shows more details about the two levels:

1. **The object level:** links the user-defined object model and the service level. We represent it as distributed OO components. Each component has an OO interface and an implementation in an OO language. The OO interface refers to a required or a provided service.

   We refer to the term `component` to describe the client or server processes as black boxes with required and provided interfaces.

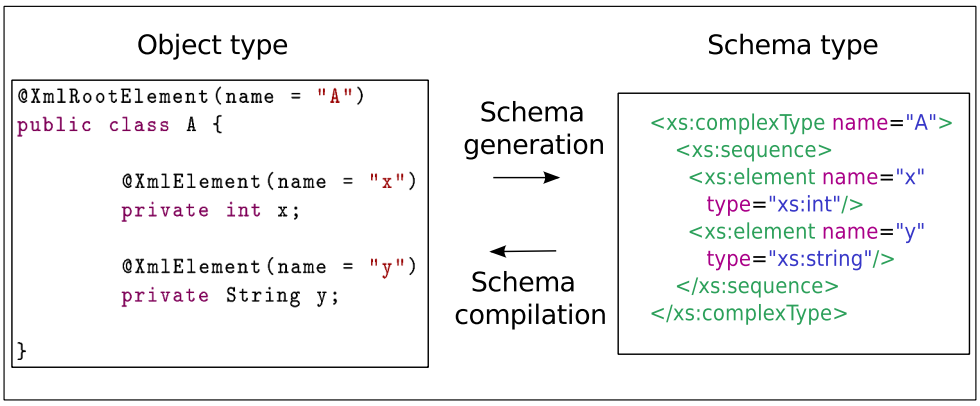


Figure 3.3: Example of schema generation and compilation

The emitted and received objects through Web services interaction are completely transparent at the object level: we represent it as a virtual object exchange in Figure 3.2.

2. **The service level:** endorses the interaction via Web services. It defines distributed structural components. Each component has a structural interface described in `WSDL` in case of a `SOAP` service or in `WADL` in case of a `RESTful` service.

   The service level enables the interaction via Web services by exchanging structural messages (like `XML` or `JSON`) between the distributed components.

Exchanges between the two levels are represented in Figure 3.2 by two functions: *i)* a `marshal` function at emission to convert objects from the object level into structures in the service level and *ii)* an `unmarshal` function at reception to convert structures from the service level into objects in the object level. These two functions are part of a `data binder`.

3.1.2 Data binder

An essential component of frameworks like `cxf` is the `data binder` [56], which binds types and values between both levels. In the object level, values are (references to) objects and types are object types, classes or interfaces. In the service level, 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 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` accepts different data binders like `JAXB` [43] (the default one) and `Aegis` for `XML`[6], or `Jettison` and `Jackson` for `JSON`[7].

Two mapping ways. The data binder binds object types and schemas representing their internal structure in a two-way mapping. The `schema compilation` produces object types

[6] http://cxf.apache.org/docs/aegis-21.html
[7] http://cxf.apache.org/docs/json-support.html

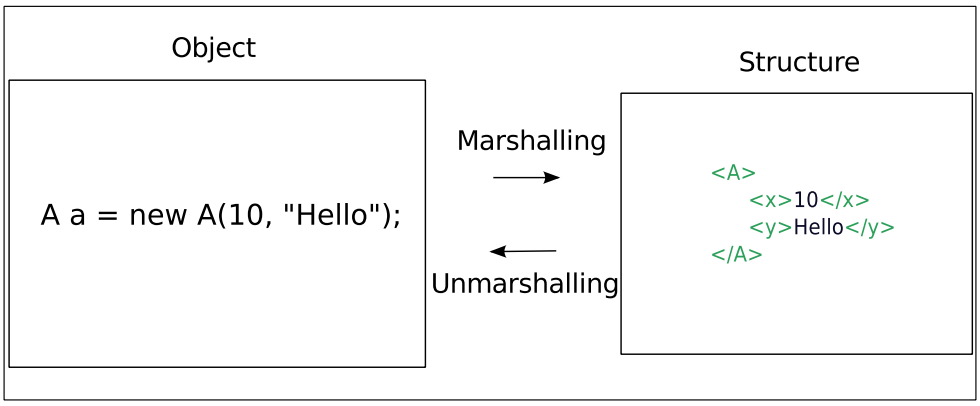


Figure 3.4: Example of marshalling and unmarshalling

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. In Figure 3.3, we show an example of a class `A` which is bound to a schema giving the name of the structure type, using `@XmlRootElement(name="A")` JAXB annotation, and the type structure as a sequence of field declarations, using the `@XmlElement` JAXB annotation.

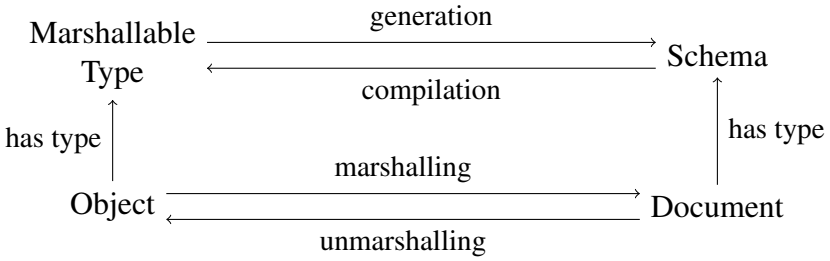
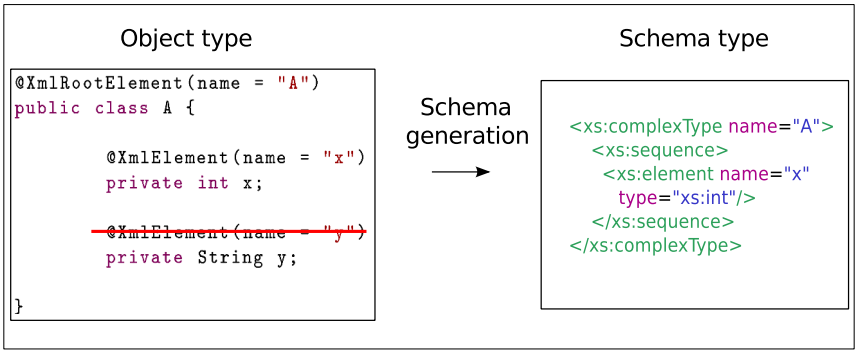


Figure 3.5: Data Binding

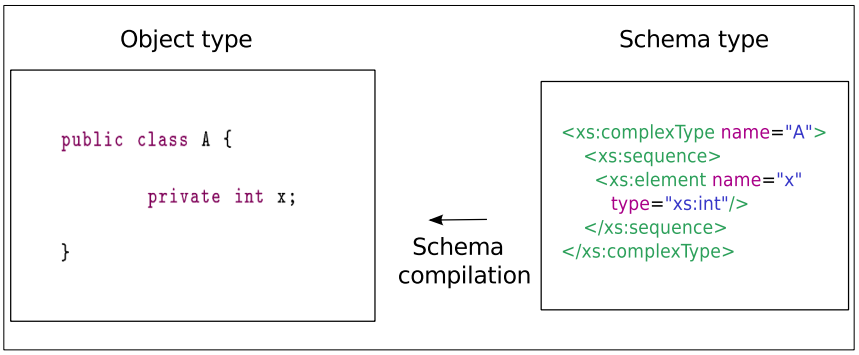
Two functions. Associated to the defined type mapping between object and schema types, two functions realize conversions in a type-safe way: the `marshalling` function maps objects to documents while the `unmarshalling` function maps documents to objects. For instance, an instance of the previous presented class `A` in Figure 3.3 is marshalled into a document labeled with name `A` and containing a marshalling of each field as a subdocument. Figure 3.4 shows the marshalling of the `A` instance followed by an unmarshalling.

Schema generation, schema compilation, marshalling and unmarshalling described previously are summarized in Figure 3.5.

Marshallable types. Note that the data binding is restricted to specific object types, the `marshallable` types. An object type is `marshallable` if it satisfies some constraints



(a) Schema generation



(b) Schema compilation

Figure 3.6: Example of irreversibility between schema generation and schema compilation

(about its constructors and fields) and defines specific mappings to its corresponding schema. These mappings are called a `binding schema` in [56]. 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 rest of this thesis, we implicitly assume that a type is marshallable, if needed.

Type equivalence. 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 [48], 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 (See Figure 3.6).

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 by an unmarshalling does not preserve the object.

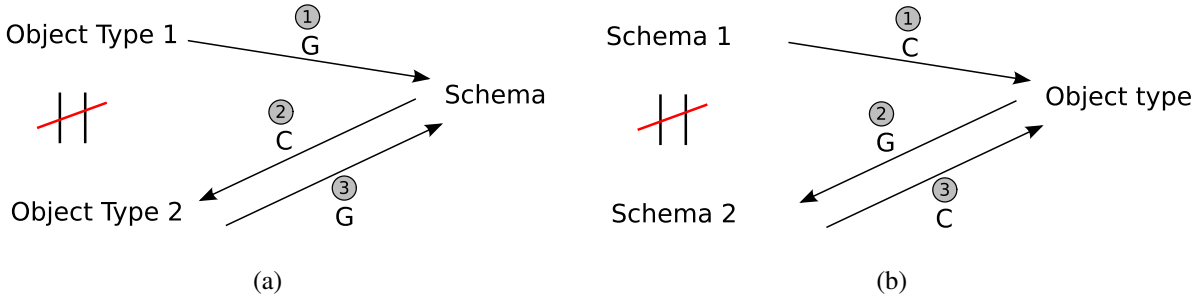


Figure 3.7: Quasi-reversibility between schema generation (represented by G symbol) and schema compilation (represented by C symbol)

However, we have observed in some data binders that the following property is satisfied, although not formally specified: the different pairs of functions are mutual quasi-inverses. Thus, starting from a schema generated from marshallable types, a schema compilation followed by a schema generation preserves the schema (see Figure 3.7(a)). Likewise in the reverse direction, starting from object types compiled from a schema, a schema generation followed by a schema compilation preserves the object types (see Figure 3.7(b)).

These properties induce a specific notion of equivalence over objects and object types respectively: it is the notion that we will use in this thesis.

Definition 1 (Equivalence for Marshallable Types) *Two marshallable objects are equivalent if the marshalling function applied to them gives equal documents, while two marshallable types are equivalent if the schema generator applied to them gives equal schemas.*

For instance, with the default data binder `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.

3.1.3 Development and execution

The section mainly explains how the framework drives the data binder, during the development phase and the execution phase [42, 31, 59].

Development. Typically, a development follows a process in two phases, located at the server and the client respectively.

1. Server side – Code-first approach
    - The developer provides some `Java` code to implement the service represented as an interface.
    - By using the object level 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`.

- The developer deploys the implementation and the contract on a server.

2. Client side – Contract-first approach

   - Using a specific tool of the object level that embeds the schema compiler of the data binding, like in `wsdl2java` or `wadl2java` tools in `cxf`, the developer produces from the contract a proxy (or also called a stub) acting as a gateway towards the server and a client skeleton.
   - The developer completes the client skeleton to produce the client invoking the service.

Execution. 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 level. Figure 3.8 describes the data flow involved in the execution of a service operation `R op(A a)`.

1. Client side – Invocation of the service

   - The client application invokes operation `op` of a service using the proxy.
   - The invocation is reified in two parts, representing a message and a channel respectively.
   - The framework calls the data binder to marshal the message.
   - The framework sends to the server the message marshalled over the channel.

2. Server side – Reception and computation

   - The framework receives from the client the message marshalled sent over the channel.
   - The framework calls the data binder to unmarshal the message.
   - From the message and the channel, the framework produces a local invocation of operation `op` of the service and calls it.
   - The implementation of operation `op` executes and possibly returns a result.
   - The framework calls the data binder to marshal the result and sends it to the client over the response channel.

3. Client side – Return

   - The framework receives from the server the response marshalled sent over the response channel.
   - The framework calls the data binder to unmarshal the response.
   - The invocation returns the response unmarshalled to the client application.

3.1.4 Data flow

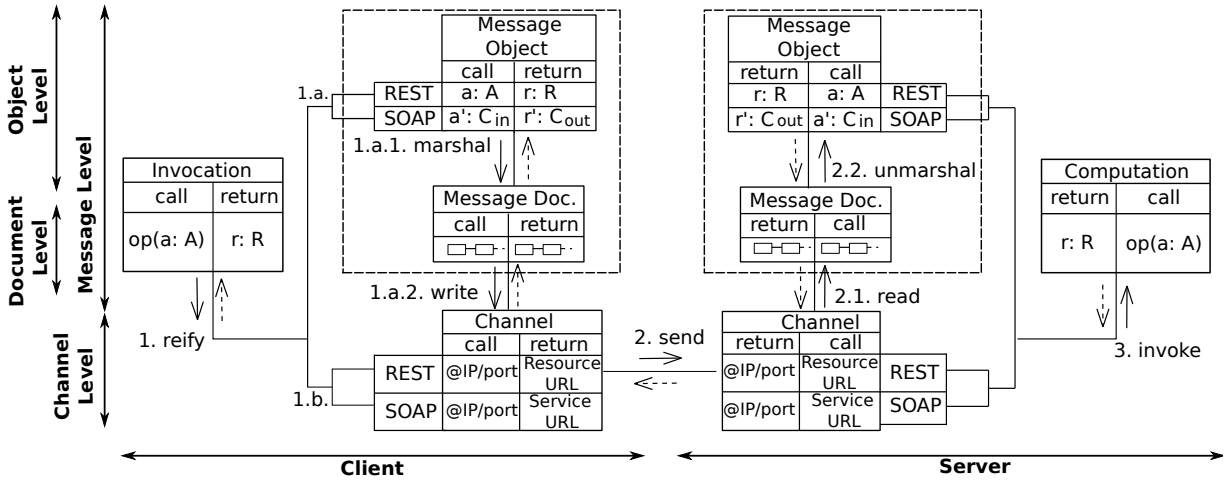


Figure 3.8: Data Flow in an OO Framework for Web Services

The main flow in Figure 3.8 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 analysis 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 3.8, the object message resulting from calling `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 3.8, 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.

3.1.5 Control flow

In the following, we are going to study in particular the control flow architecture of the `cxf` framework. Based on Apache `cxf` software architecture guide[8], the overall `cxf` architecture is primarily made up of the following parts:
(1) Bus: it is the backbone of the `cxf` architecture. It defines a common context for all service endpoints. It wires all the runtime infrastructure components and provides a common application

[8] http://cxf.apache.org/docs/cxf-architecture.html

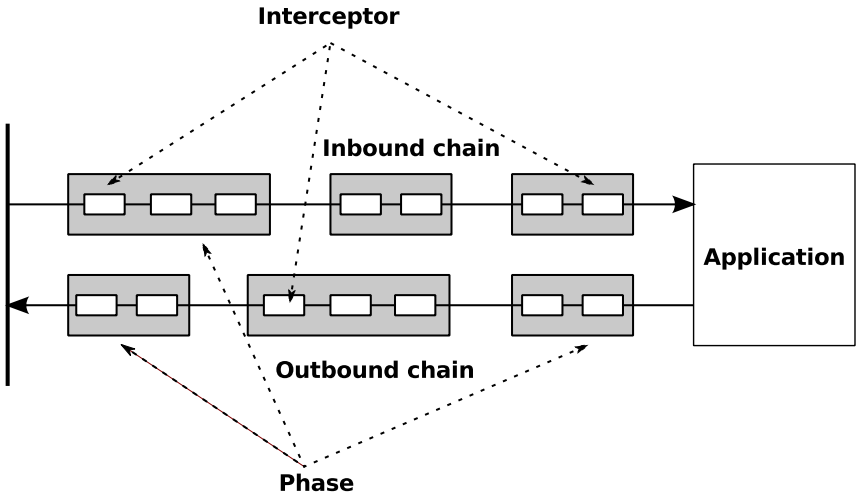


Figure 3.9: Inbound and outbound chains in cxf

context,
(2) Front-end: Front-ends provide a programming model to create services, like JAX-WS and JAX-RS,
(3) Messaging & Interceptors: They provide the low level message and pipeline layer upon which most functionality are built,
(4) Service Model: Services host a service model which is a WSDL-like model that describes the service. The cxf front-ends internally use the service model to create Web services,
(5) Data Bindings plug: it is an interface to plug data binding frameworks like JAXB, Aegis and others,
(6) Protocol binding: it binds the Web services messages with the protocol specific format. cxf supports the following bindings: SOAP, CORBA, Pure XML,
(7) Transports: Transports define the high-level routing protocol to transit the messages over the wire. cxf supports the following transports for its endpoints: http, CORBA, JMS, Local.

A lot of details exists in such an architecture. In the following, we zoom only on the part which concerns the marshalling and unmarshalling processes by focusing on interceptors.

Interceptors are used with both cxf clients and cxf servers. When a cxf client invokes a cxf server, there is an outgoing interceptor chain for the client and an incoming chain for the server. When the server sends the response back to the client, there is an outgoing chain for the server and an incoming one for the client. Additionally, in the case of errors, cxf will create a separate outbound error handling chain and the client will create an inbound error handling chain. Interceptor chains are divided up into phases[9]. Each phase has a particular role on the service invocation. Interceptors within a phase are organized sequentially in the order of execution (see Figure 3.9). We are interested here in the Unmarshal phase in the incoming chain and in the Marshal phase in the outgoing chain. The interceptor in the Unmarshal phase will create a message Reader which has a specific unmarshalling type. The unmarshalling type is

[9]For an access to the complete list of phases, please refer to the following page from the cxf guide https://cxf.apache.org/docs/interceptors.html

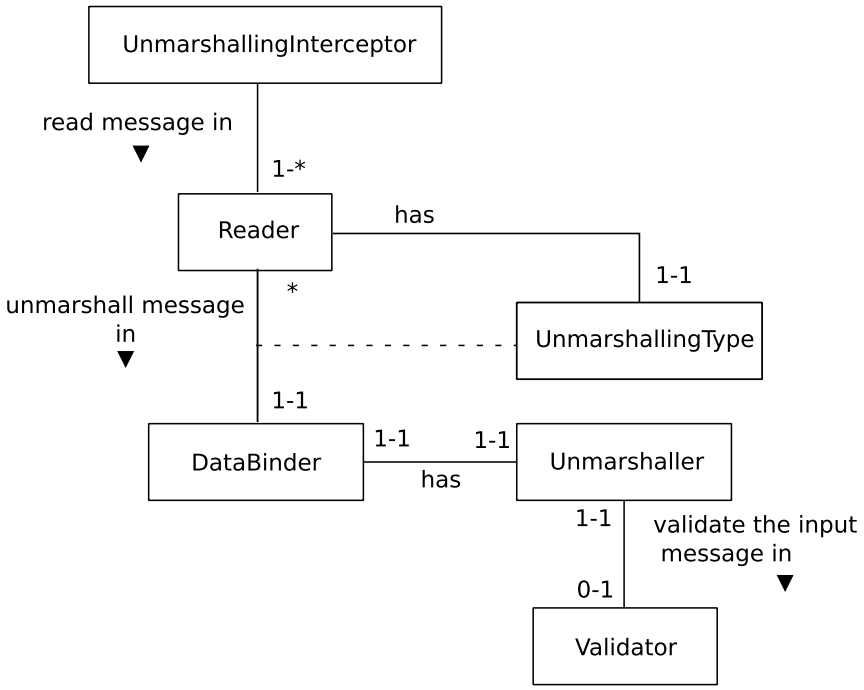


Figure 3.10: Abstract UML of the unmarshalling phase in `cxf`

determined statically, from the declaration of the service. The `Reader` uses the unmarshalling type to call the unmarshal method on a data binder in order to transform a document into an object. The unmarshalling is achieved by the unmarshaller part of the data binder. The `Reader` can ask the data binder to validate the input message following its structural type before unmarshalling. In this case, the `Reader` must set a validator to the unmarshaller before proceeding for unmarshalling.

Figure 3.10 shows an abstract UML class diagram of the unmarshalling phase. In a similar manner, the interceptor in the `Marshal` phase will create a message `Writer` which has a specific marshalling type. The marshalling type could be the dynamic type of the object, determined in `Java` by a call to method `getClass`, or its static type, coming from the declaration of the object marshalled. The `Writer` uses the marshalling type to call the marshal method on a data binder in order to transform an object into a document. The marshalling is achieved by the marshaller part of the data binder. The `Writer` can ask the data binder to validate the output message following its structural type after marshalling. In this case, the `Writer` must set a validator to the marshaller before proceeding for marshalling.

Figure 3.11 shows an abstract UML class diagram of the marshalling phase. For more technical details about the implementation classes in `cxf` matching with the UML diagrams of Figures 3.10 and 3.11, please refer to Appendix A.

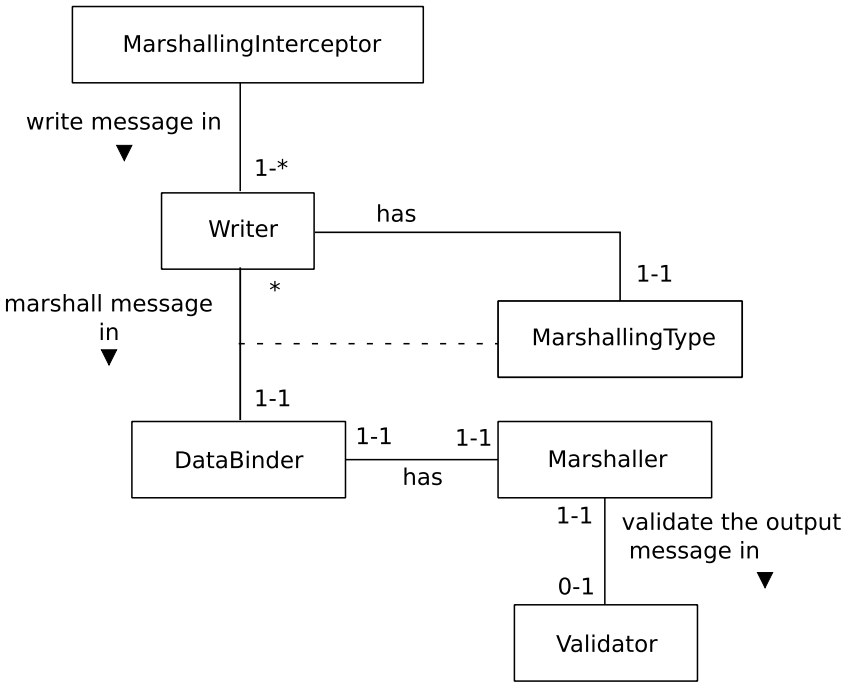


Figure 3.11: Abstract UML of the marshalling phase in `cxf`

3.2 An overview about distributed object environments

In this section, we discuss two axes in the distributed object environments. First, we present the principle engineering designs in such environments. Second, we quote some principles in the OO middleware for implementation and deployment in distributed object environments.

3.2.1 Distributed objects design

In the following, we mention some object design properties in distributed objects [25].

Life cycle. The distributed object life cycle considers object creation, migration, activation/deactivation and deletion:

- Object creation: Objects on a host are capable of creating objects elsewhere. Object creation must respect the principle of location transparency: where to create objects should be determined in such a way that neither client nor server objects have to be changed when a different host is designed to serve new objects.

- Object migration: If a host becomes overloaded or needs to be taken out of service, objects hosted by that machine need to be moved to a new host. Migration has to address machine heterogeneity in hardware, operating systems and programming languages.

- Object activation/deactivation: Sometimes, hosts have to be shut down and then objects hosted on these machines have to be stopped and re-started when the host resumes operation. Moreover, depending on the nature of the client application, objects may be idle for a long time and it would be a waste of resources if they were kept in virtual memory all the time. For these reasons, designing distributed objects must consider activation/deactivation of objects. Activation launches a previously inactive object, brings its implementation into virtual memory and then enables it to serve object requests. Deactivation is the reverse operation: it terminates execution of the object and frees the resources that the object currently occupies.

- Object deletion: Objects in a non-distributed application may be deleted implicitly by garbage collection techniques, which are available for instance in `Java`. The garbage collection technique is complex in distributed environments since it would require that objects know how many other distributed objects have references to them. Most distributed object systems do not fully guarantee referential integrity due to its expensive achievement. Consequently, the client objects have to be able to cope with the situation that their server objects are not available any more.

Object references. References to distributed objects are data structures. They need to encode location information, security information and data about the type of objects. Due to this consideration, designing distributed object-based applications must minimize the number of objects. Indeed, applications cannot maintain a large number of object references since they would demand too much virtual memory on the client side. Moreover, the distributed middleware has to know all object references and must map them to the respective server objects. Therefore, the distributed object design must choose the granularity of objects such that both clients and middleware can cope with the space implications of object references.

Request Latency. In order to optimize the performance of an object request in distributed environments, the implemented-code design must make assumptions about the overall cost of a simple object request, the size of the representation of an object reference and the increase of latency depending on the size of the transmitted information.

Parallelism. The client object that requests a service always executes in parallel with the server object. If it requests services in parallel from several server objects that are on different hosts, these requests can all be executed in parallel. Therefore, in order to avoid integrity violation in a concurrent execution of two operations, such as inconsistent analysis or lost updates, it is required to implement concurrency control within the server objects.

Communication. Distributed object communication primitives need to be designed that facilitate communication between groups of objects. In order to reduce the request latency, it is recommended for the distributed object design to process multiple requests (to one or several objects) at once.

Failures. Distributed objects have to be designed in such a way that they cope with failures. The fact that request may fail, hosts must check for exceptions that occurred while the request was executed. Sometimes, a host may have a sequence of requests that should be done either completely or not at all. Therefore, distributed objects must be built in such a way that they can participate in transactions. This allows to make provisions to undo the effect of changes.

3.2.2 Object-oriented middleware

OO middleware evolved from the idea of remote procedure calls (RPC). The first of these systems was the OMG's Common Object Request Broker Architecture (`CORBA`) [81], then Sun provided a mechanism for Remote Method Invocation (`RMI`) [83] in `Java`. `RMI` needs the same language at the client and the server side (`Java`) while `CORBA` allows an heterogeneous approach. They differ in the communication protocols and in the way to serialize and deserialize the information. `RMI` and `CORBA` use specific protocols (`RMI` and `IIOP` protocols).

In the following, we present some principles of OO middleware [26].

Interface Definition Language (IDL). OO middleware provides interface definition language (IDL), where interfaces define object types and instances of these types are objects in their own right. Interfaces can be seen as contracts that govern the interaction between client and server. They are also the basis for distributing type information. Several modern programming languages, such as `Java`, provide a mechanism for interfaces. These programming language interfaces should not be confused with the interfaces for distributed objects that are written using an IDL in a way that is independent of programming languages. Programming language interfaces are a particularly useful vehicle for the type safe implementation of server objects. Following this approach, the IDL compiler generates a programming language interface that includes all operations exported by the IDL interface.

Marshalling/Unmarshalling. OO middleware supports client and server stubs, which perform marshalling/unmarshalling and resolve heterogeneity of data representation. Client and server stubs are proxies for servers and clients. They are automatically derived from interfaces and generated by the IDL complier that is provided by the middleware. Figure 3.12 shows the role of the generated stubs (at emission and at reception) to perform object request through the transport layer, in order to mimic the local method calls in an OO environment.

At the transport layer, objects need to be marshalled and unmarshalled in a similar way to data structures. These data structures are representations of object references in a sequence of bytes: the marshal operation transforms objects into a sequence of bytes and the unmarshal operation performs the reverse operation.

Inheritance. The implementation of server objects could be directly done by the implementation of interfaces of the programming language corresponding to the IDL or by inheritance. For server implementation with inheritance, the middleware generates an abstract class as part of the server stub. This abstract class has abstract operations that declare the operation signature

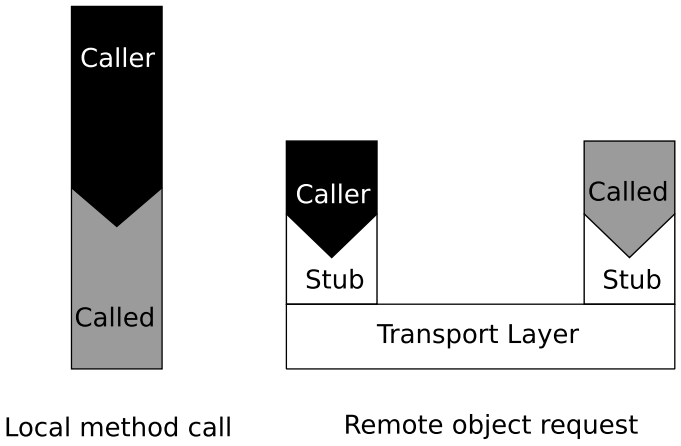


Figure 3.12: Local method calls versus remote object requests [26]

but are not implemented by the class. The class implementing the server object is then defined as a subtype of this abstract class. The server stub will have a static reference to the abstract class but it will refer dynamically to an instance of the server object. The programming language compiler used for the server implementation will check that the redefinition of abstract operation meets the declaration of the respective operations in the abstract superclass.

Connection between multiple objects. The OO middleware provides the connection between multiple objects over one or several connections established by the transport layer. In more concrete terms that corresponds to mapping object reference to hosts, activation and deactivation of objects, invocation of the requested operation and the synchronization of client and server. The whole principle is summarized in Figure 3.13. First based on the object references which identify a host, the middleware finds the host on which to execute the request. Then, it contacts an object adapter on that host which locates or activates the process in which the object resides. The server process then identifies the object implementation on which the requested operation is executed.

Server registation. Object activation might necessitate the object adapter to start a server object implementation. Once the server objects have been compiled, they need to be registered with the OO middleware. The purpose of this registration is to tell the object adapter where to find the executable code for the object and how to start it. The registration is generally done by a system administrator, who maintains an implementation repository. The implementation repository associates objects, which are usually identified by an object reference, with some instructions. There is an implementation repository on each host.

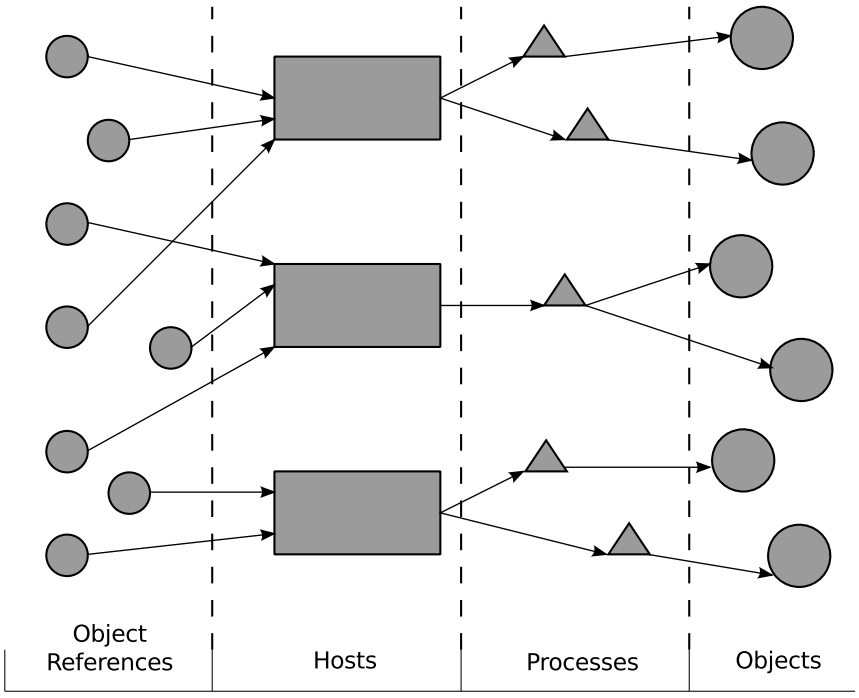


Figure 3.13: Distributed objects: Mapping the received object references to local objects [26]

3.3 Discussion

In the following, we discuss first the existing mismatch between the service orientation and the object orientation in distributed systems, by considering particularly the subtyping concept. Then, we discuss the mismatch between objects and structures. We finish by introducing a need for a unified model at the service level.

3.3.1 Service orientation vs object orientation in distributed systems

Web services support a document-oriented style for clients to interact with a server. This technology has been criticized by several authors which compare it with `CORBA`, `RMI`, and other well-established approaches for distributed systems. For instance, [82] argues that Web services are not distributed objects since they mainly lack object references and lifecycle management. There are mainly three standards for interoperating between `Java` distributed applications: `RMI`, `CORBA` and Web services. All three kinds rely on common ideas: a service language (interface, IDL, `WSDL`), a notion of service discovery and a mean to call such a service as we previously described. There is always a significant debate about the merits of Web services compare to prior technologies. There are currently several comparisons between the three concurrent middleware technologies [30, 32, 41, 22] but mainly they are performances comparisons. These performance or "non functional" services are out of our interest for this thesis. We more particularly focus on subtyping in object-oriented distributed systems. Indulska [39] considers the `substitution principle` as a key requirement. The `substitution`

principle [51] would allow different substitutions to take place in the object level: 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.

Subtyping in the existing OO middelwares. 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. RMI allows objects and classes to be sent to the remote server when the needed subtype is not known at reception.

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 subtyping relation that can be used to ensure substitutability. However, in CORBA, objects are passed by references: there is no translation of objects. The substitution principle can be fixed with CORBA using the dynamic skeleton interface and the dynamic interface invocation facilities.

Web services subtyping. Subtyping is not respected in the existing Web services development and deployment practices. Clients calling a Web service must get its structural service interface and generate locally the corresponding classes. Therefore, classes of the development environment are twinned between the client and the server. This similarity ensures a successful interoperability for Web services but a tight coupling between clients and servers.

Thus an open question is raised here:

How to improve Web services interoperability with subtyping in a consistent way?

3.3.2 Gap between objects and structural documents

Data binding is a way to interoperate between two different formats, usually XML. But there is neither a precise definition nor a previous theory for this concept. However, there are many frameworks supporting it and XML data binding is rather a popular technology with JAXB for Java. But solutions also exist with C++ and Python [56]. In the data binding, XML is surely well-known but competing languages are YAML and JSON. There are also data bindings for YAML and JSON. We rather consider the communication layer as transparent in our work. It could be JSON, XML or any other format.

The mismatch between XML and OO technologies is a well-known domain [48], nevertheless solutions to fill the gap are rare. The approach in [3], provides a core subset of XSD and an algorithm to convert XML documents into objects and vise-versa. This is a point of view starting from an XML document and going to an object representation and back to an XML document. As far as we know, there is no study handling the reverse point of view starting from an object representation and going to a document and back to an object, more particularly in the context of preserving the OO subtyping in the converted documents. Such a subtyping preserving principle allows for instance to process successfully the following exchange for instance: an instance of type B is marshalled at emission into a document doc_B and then unmarshalled at reception as an A instance, supertype of B. Often, using the existing data binding frameworks like JAXB,

doc_B could not be matched as a valid document of type A when it is missing the subtyping relation between B and A in the received document. Another approach to reconciliate objects and XML structure is to embed the structures as objects in the language. One such approach is [45] which defines XML objects from XML schema and makes them first-class data values. The authors define a Java extension (called XOBE) where XML syntax denotes XML objects and whose validity is mainly checked at compile time. Related approaches are Xduce[10] and XTatic[11]. We are not directly concerned with the comparison of XML schemes as in [50] and other similar work.

3.3.3 A need for a unified model at the service level

In the previous chapter, we have deduced that there is a need to bring the dynamic discovery practices with subtyping to the existing Web services development environments (following the "Updating service access methodology" for dynamic discovery). That could be by facilitating and unifying the accessibility to the existing discovery protocols. In this chapter, we have deduced that, in distributed object environments, there is no concrete standardization to enable a safe OO development of Web services by considering subtyping.

By combining these two requirements, we deduce that in an OO environment for Web services development, there is a need for a framework which:

- is transparent from some complex details at the service level in order to allow a uniform and standard technique for Web services discovery independently from the service model and from the used discovery protocol,
- allows a safe interaction by considering OO subtyping on required and provided interfaces.

In order to achieve this goal, there is a need to conceptually unify the concepts at the service level in order to offer development facilities at the object level.

We are talking about an interaction model, or also a unified black box model where a service is represented as a black agent implementation with a structural interface (WSDL or WADL). Figure 3.14(a) represents such a model: the black circle wrapped with a pointed square is the implementation and the arc is the structural interface. When the Web service is implemented using an OO framework, the agent implementation is represented as a black circle with its OO interface the whole wrapped with the pointed square, as it is represented in Figure 3.14(b).

The advantage of such a black-box paradigm of distributed systems is to dissociate the agent implementation from its interface. The idea is that:

- a Web service could keep the same implementation and change the structural interface which implies changing the Web service model,
- a Web service could keep the interface (the Web service model) and change the implementation.

[10] http://xduce.sourceforge.net/
[11] http://www.cis.upenn.edu/~bcpierce/xtatic/

(a) Pur structural Web services interaction

(b) Web services interaction using an OO implementation

Figure 3.14: Distributed black boxes abstraction

These two `black-box` properties imply two important points in the context of an OO Web service framework:

- Loose coupling between the object and the service levels
- The OO interface and its corresponding structural one must be adapted in the sense that messages are safely exchanged through and by these two interfaces (see Figure 3.14(b)). Such a property implies that the subtyping relation at the OO interface must be hold by the structural one in order to preserve interoperability between distributed agents.

Therefore, there is a need to define a typed unified model for `SOA`. Such a unified typed model helps as a basic reference:

- first, it is useful as a formalization of the service level with an expressive and sound type system with subtyping,
- second, it is useful to prove that the two Web service models, `RESTful` and `SOAP` could be conceptually unified despite their differences,
- third, it helps as a foundation for an OO Web service API for the development of dynamic discovery (client side), based on the unified concepts, far from the technical details at the service level,

- fourth, the type system of the unified model will help to fix the existing gap between the object interface and its corresponding structural one in the meaning of ensuring interoperability by subtyping.

In the next chapter, we will discuss the existing formal models of service-oriented computing which mainly support dynamic discovery and subtyping.

4

Abstract Formal Models for Service-Oriented Computing

Contents

SOA is based on services as distributed components. These components are mainly based on service description interfaces. These interfaces are often known as structural standardized interfaces like WSDL for SOAP and WADL for RESTful. In this chapter, we present a state-of-the-art of abstract formal models based on messages exchange between distributed components.

First in Section 4.1, for modeling Web services interaction, we focus on the state of the art of modeling the communications between local executions of services and the network. Compo-

nents are seen as distributed black-boxes with interfaces. We focus also on the dynamic discovery which makes the network topology evolve. We consider here the "Updating service access methodology" for dynamic discovery (see Section 2.1.2). Therefore, there is no need to regenerate the service interface dynamically, (only the service reference address is updated). We are also interested in work comparing between the two models, SOAP and RESTful. Then in Section 4.2, in order to unify the type of exchanged messages in SOA which support contents for service discovery, we discuss a well known formal type system. This type system supports dynamic discovery with type inference and subtyping. We compare it to the existing used type systems for Web services to show their weaknesses. Finally in Section 4.3, we discuss the existing work for type safety and type checking.

4.1 Abstract models for SOA

In this section, we present first the communication requirements in SOA. Then, we present an overview about the existing message-based models for SOA. Then, we discuss few work to integrate Web services models. Finally, we focus on π-calculus and Join-calculus as two standard models for modeling interactions between processes. We discuss why Join-calculus is more adapted to distributed system than π-calculus.

4.1.1 Communication requirements

The communication model for service-oriented computing must have the following characteristics: Synchronization, message-passing model and mobility which we detail in the following.

Synchronization. There are two kinds of synchronization in distributed systems: (i) Synchronous communication, which is defined as a communication where the sender remains sleepy waiting for its request to be accepted. (ii) Asynchronous communication, where the sender sends a request and continues its execution without waiting for an answer. It is possible to implement asynchronous communication over a synchronous execution level and vice-versa by specifying suitable communication protocols [20]. When an asynchronous model is used on a synchronous execution level, buffers can be used to store messages and allow the sender to continue its execution. While when a synchronous communication is modeled over an asynchronous system, the sender remains blocked waiting for an acknowledgment message from the receiver.

Service-oriented computing is often represented as asynchronous communications which provide a flexible communication mechanism and allow also to simulate synchronous communications.

Message-passing model. Following Lamport and Lynch's criteria [49], in a message-passing model components are black boxes, send and receive messages by using a buffer (the network) and without sharing memory or without synchronizing the sending and the receiving of messages in a rendez-vous. Message-passing models abstract from the details of communication.

Mobility. Mobility in service-oriented computing allows dynamic binding which is necessary for service discovery. Indeed, during an execution, the network topology often needs to evolve: an agent needs to discover another agent that it does not know initially.

4.1.2 Message-based models

A wide variety of formal models exists for service-oriented computing. Two distinguished approaches of formalization are presented: (i) process calculus models for expressing and analyzing service based-systems, [80] or (ii) models for giving a formal semantic for a standard orchestration language, like BPEL, [52]. The drawback of these orchestration models is that they present a service implementation formalism for local processes description, which complicates the model with multiple communication rules. These models are out of our interest for this thesis. We require a simple formal model hiding the local details which make it independent from implementation languages used for describing processes execution which are more or less known as black box models.

There are some interesting work for the black-box principle, like [74, 44, 71]. Seehusen and Stolen in [74] define a formal and abstract model for services. The semantics are based on a notion of trace which is a sequence of events. An event is either a transmission (!) or a receipt (?) of a message compound (emitter, receiver, content). The intent of this formalization is to abstract message sequence chart as they are used in UML 2. While in [44], authors use the notion of abstract state space to specify the functional descriptions of Web services. The proposed model defines formal semantics abstracting from the concrete underlying language. A Web service is defined as a total function mapping input values and states to the execution traces. The authors demonstrate the applicability of the formal model by showing how to define and determine realizability and semantic refinement. These work have different abstraction interests: [74] presents an advanced formalization of some confidentiality issues and [44] focuses on a particular functional description without a clear defined syntax.

4.1.3 Unified models for SOAP and RESTful services

The problem of the integration of SOAP and RESTful services has received some attention in industry[1]. There are few studies related to composition and integration of heterogeneous services in the academic realm. An exception is [36] which compares different techniques and proposes an hybrid orchestration approach. Practically their solution is to design sub-workflows dedicated to manage each technology in a native way and to minimize the interactions between both parts. The work from [6] addresses interoperability issues in web, grid and p2p services, but it does not discuss RESTful services. The authors advocate for a generic and abstract conceptual model based on the notions of service and messages. A last approach is to provide a transformation of one technology into the other. This is, for instance, the purpose of [76] which defines rules to automatically convert a SOAP architecture into a RESTful one.

Concerning work about interfaces, Carpineti in [17] provides a notion of service types that includes a notion of typed channels and set operators over types. This work puts a theoretical

[1] http://www-01.ibm.com/software/webservers/smash/#, http://ode.apache.org/

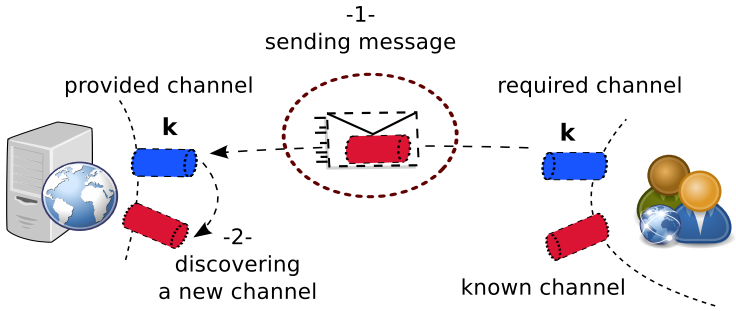


Figure 4.1: A client/server communication with a channel discovery

foundation for Web services interfaces. We discuss this type system and compare it to a more complex one in the the rest of this chapter.

4.1.4 From π-calculus to Join-calculus

π-calculus [63, 58, 70] is a classic process calculus which is used to formally specify and verify concurrent systems. It is a model based on atomic interactions (Rendez-vous) and requires that the sender and the receiver synchronize at specific interaction points, where they can exchange information synchronously (it is still possible to use π-calculus to model asynchronous [16] interactions but it is less known for such a use). A system is represented as a set of independent agents or processes. Interaction between processes is modeled using channels. A channel is an abstraction of the communication media on which data is exchanged. π-calculus allows also the transmission of channels as a content in exchanged messages to be discovered at reception. An illustration about this principle is given in Figure 4.1, where we represent a client and a server communication. The client requires the channel k provided by the server. The sent message on k contains a channel which is discovered at reception by the server and added to its interface.

However, π-calculus is more adapted to local service orchestrations than distributed systems, because it lakes the notion of locality on exchanged channels [28]. In order to explain the locality principle, we consider the following example of an agent that can receive a channel then add rules on this channel. For instance, we consider a client, let us call it c, which receives a channel l on an input channel k and then uses l to receive a data x before processing as P: $k(l).l(x).P$. This example is inconsistent with mobility and distribution because the discovered l channel must have a unique location (URI) at a distinct agent, let us call it p, such that all sent messages on l are received by p and only p. Therefore, l could not be used as a local channel to receive messages on c.

In order to resolve this problem, Fournet [28] proposes a reflexive chemical machine associated to the Join-calculus which extends the Berry's chemical abstract machine [11] (CHAM) with the notion of locality and reflexion:

- CHAM: It brings a semantical behavior to π-calculus. It describes the system as a chemical solution, where floating molecules interact with each other, producing new

molecules, according to reaction rules. Other rules, called parallel rules, decompose molecules into smaller molecules, or to compose bigger molecules from smaller ones. The effect of these rules, contrary to reaction rules, is reversible.

- Locality: Molecules travel directly to the location where they will react and where pattern matching is applied only to channel name. Each reaction rule or molecule can be associated with a single reaction site.
- Reflexion: It allows reactions to extend the machine with new kinds of molecules with their reaction rules.

4.2 Type systems

In this section, we present an overview of the state of the art for type systems of services belonging to an interaction model. The principles of this interaction model were defined in 3.3.3. Channel mobility in such a model represents the discovery of a service location. The type of the discovered service is defined at compile time. Therefore, at execution, the type of the received channel is inferred. This mechanism is known as `type inference` [75].

First, we present the expressivity needed requirement in such an `interaction model` with channel mobility. Second, we present the type systems in theory and the weaknesses of the existing type systems conforming to this theory. Finally, we discuss type safety and type checking work.

4.2.1 Typing requirements for an interaction model

We present here some requirements in a Web services communication which lead to an expressive syntax for Web services interaction.

4.2.1.1 Channels mobility: channel type and recursivity over channel types

Channel mobility has different application contexts:

Request/Reply. In an asynchronous service model, a client must communicate a reply channel to its server when making a request. At reception, the server must infer this reply channel to its expected return type. Then, it will use it to reply to the client. Thus, when a service reply is expected, an input channel at the server must define a type that contains a return channel type used to infer the type of the received channel from the client.

Service discovery. Following the same principle as for the request/reply case, service discovery involves channels transmission. At reception, the Web services component discovers the new channel and infers from it the expected type. This discovered channel is used to invoke a remote service. Thus, an input channel, where a channel discovery is expected, must define a type that contains a discovery channel type used to infer the type of the received channel.

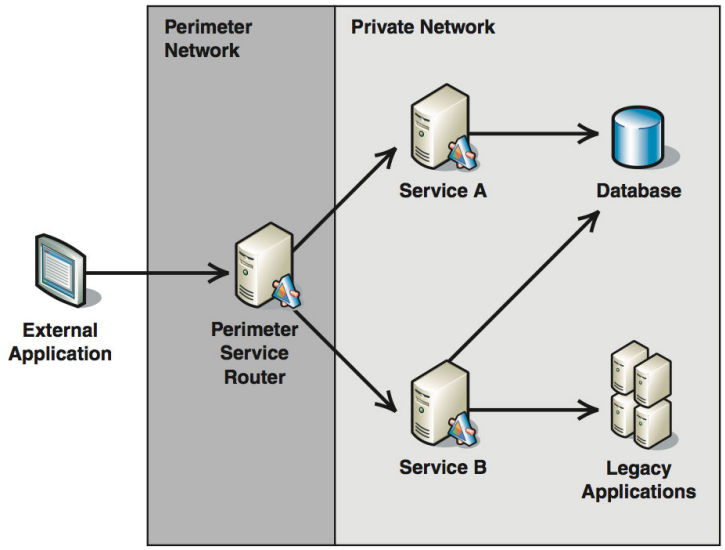


Figure 4.2: A perimeter service router on the perimeter network [23]

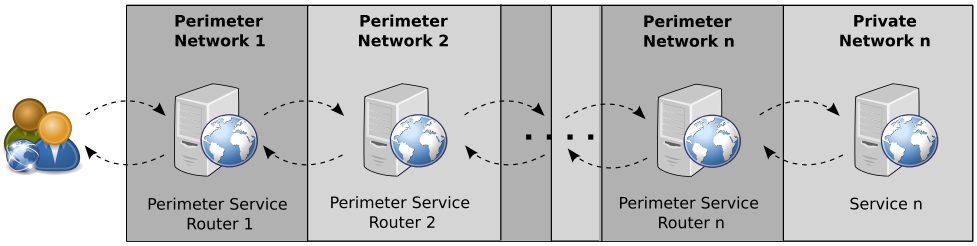


Figure 4.3: Web services routing through a chain of private networks

Web services routing. For security reasons [23], Web services in a private network could not be directly accessible to external clients. For that, a perimeter service router is needed. Figure 4.2 illustrates such an architecture. The perimeter service router provides an external interface on the perimeter network for internal Web services. It accepts messages from external applications and routes them to the appropriate Web service on the private network. In this figure, we distinguish three parts:

- **External application:** An application located outside of the private network that needs to access the Web services in a private network.
- **Perimeter service router:** The perimeter service router is a Web service that provides access to Web services in the private network.
- **Service:** One or more Web services that are accessed by the perimeter service router.

4.2.1.2 Succession of interactions

Sometimes, calling a Web service could require a succession of message exchanges before reaching the wanted destination. In the following we provide two examples about such an interaction need.

A chain of Web services routing. The service routing previously presented could be done through a set of multiple routing and rerouting of messages before reaching the target "service n" in the "Perimeter Network n". In Figure 4.3, we show a client and a chain of "n" perimeter service routers. We assume a variable $i/1 < i < n$, when perimeter service router i receives the message from the client, it must identify a continuation channel for the rest of the chain which must add additional treatments on the message. This continuation must be a channel of "perimeter service router i+1". When reaching the "perimeter service router n", the message has one last destination, "service n" in the private network zone of this perimeter service router. "service n" applies the final treatment on the message. The message reply will cross the inverse path of the request message.

According to this definition, the type of the input channel of "Perimeter Service Router 1" is the type of the whole chain, while for the "Perimeter Service Router 2" input channel, it is the type of the chain starting by "Perimeter Service Router 2" and so one. For such an example, it will be hard to specify exactly the type of different channels because either the length of the chain could be unknown or it is known but very long and so hard to be expressed. Thus, there is a need to define a recursive type on channels in order to hold such kind of Web services routing.

Such a chain type is useful also in case of peer to peer networks where an agent interacts with the closest neighbor in order to reach the final destination.

Multiple client/server exchanges. In order to establish a connexion between a client and a server, sometimes it is required to establish a number of valid exchanges. Such kind of interactions could be useful to orchestrate services (like BPEL [60]), to secure services (like OAuth [35]) or simply to organize the access to services (like RESTful self-discovery principle by Linked Data).

4.2.1.3 Customization of Web services interfaces

When a service provider describes the service interface, he may want to customize input and output types for exchanged data and channels. In order to enrich the type system to support such customizations, we need three more constructors: Negation, Union and Intersection. The use of these constructors has been proved useful for processing semistructured data in a query language, NoSQL [9].

For instance, for a flight reservation service, a reservation may be defined using a Record type which specifies the departure city, the arrival city and the date of the flight. The service provider wishes to not handle reservations from and to middle east. For such kind of reservations, we specify a type MERecord which is a subtype of Record. Thus, the Web service

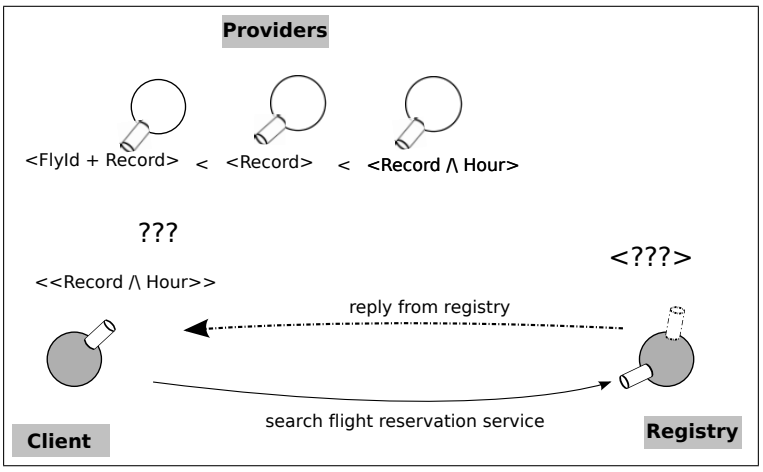


Figure 4.4: Searching a flight reservation service.

channel type must specify a negation on values of subtype `MERecord`.

4.2.1.4 Subtyping

Subtyping of services is useful, in particular, in order to type check messages and to enable services to be provided by more specific ones.

In the following we present an example about the utility of subtyping. Let us consider an example in the context of a flight reservation system. A reservation may be defined using several forms: (i) a flight identifier `FlyId` or a `Record` defining the flight date as well as the departure and arrival cities, (ii) only a `Record`, or (iii) a `Record` and an additional time specification `Hour`. These types can be respectively typed using general set-theoretic operators on types as follows: `<FlyId+Record>`, `<Record>`, `<Record ∧ Hour>` (in these examples, angular brackets `<t>` denote service channels that convey messages of type `t`). A client may call one of these services, *e.g.,* by sending one of the following messages: `(AF377)`, `(dep(Nantes), arr(Paris), 06/12/2012)` or `(dep(Nantes), arr(Paris), date(06/12/2012), 9:30PM)`.

Let us now consider service discovery, an important part of `SOA` where services dynamically appear and disappear. Service discovery also allows a user to dynamically search a given service, generally via some specialized web site called a registry, see Figure 4.4.

A client may, *e.g.,* declare interest in services of type `<Record ∧ Hour>`. This means that he should declare the reception channel as `<<Record ∧ Hour>>`, (it is the type of a channel which waits the receipt of a channel with type `<Record ∧ Hour>`). Then he can query a registry to get a flight reservation service. Depending on the provided information, the registry can send to the client various services that must be compliant to the client's expectation. Since channels, analogously to functions, have to be contravariantly typed, the following subtyping relations hold: `<FlyId+Record>` $\leq$ `<Record>` $\leq$ `<Record ∧ Hour>`. In our example, the reception of services with any of the three types `<Record ∧ Hour>`, `<Record>` and

<FlyId+Record> is correct (*i.e.,* is compatible with the client specification). This service should be sent by the registry via a channel, which requires subtype relations between channels to be defined (similarly to interface compatibility in component based systems, see for instance [72]). Compatibility means that a provided service has a subtype of the required service. Thus in our example, <<Record>> is correct for the output channel of the registry: it constraints the transmitted values to be channels of type either <Record> or <FlyId+Record>. As this example shows, compatibility rules for typed services are quite intricate, notably in the presence of expressive value types and typed first-class channels; automatic type checking and a corresponding soundness proof are needed.

In the following, we discuss the existing used and theoretical type systems conformally to these requirements.

4.2.2 Type systems in theory

We are interested in type systems that satisfy the previous typing requireement. The state-of-the-art academic type system of Carpineti and Laneve [17] is much more expressive than WSDL-based typing with its typed service endpoints. Carpineti's system notably provides a notion of service types that includes notions of typed channels and set operators over types. It is, however, less powerful and regular than so-called semantic type systems that have been introduced for programming languages by Castagna [29] and that we harness in the present work. Indeed, Castagna type system matches with the typing requirements for service interactions with channel mobility and subtyping. It extends a type system with set operators:

$$t \ ::= \ \mathbf{0} \mid \mathbf{1} \mid t \to t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

0 and **1** respectively correspond to the empty and universal types, $t \to t$ is the function type, $t \times t$ is the product type, $\neg t$ is the negation type, $t \vee t$ is the union type and $t \wedge t$ is the intersection type. This type system is expressive and powerful because it allows to represent channels as a particular function type because a channel corresponds to a service function, it allows to customize Web services interfaces using $\wedge, \vee$ and $\neg$ type constructors. Moreover, using recursivity, this type system allows the representation of complex chains and Web services routing (more concrete examples will be presented in Chapter 6). Several ways are possible to formalize recursive types in their formal type system: *i)* introduce them with explicit binders $\mu x.t[x]$, or *ii)* define them as regular trees generated by their grammar, or *iii)* define them as the solution of systems of equations. In order to ensure the definition of a constructive recursive type, they require that every infinite branch has infinitely many occurrences of the $\times$ or the $\to$ constructors.

Moreover, Castagna in [29] proposes a semantic subtyping naturally interpreted as set inclusion. He considered an interpretation of types as sets of values:

$$[\![t]\!]_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

The subtyping relation between two types s and t is basically defined as follows:

$$s \leq t \iff [\![s]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}} \iff [\![s \wedge \neg t]\!]_{\mathcal{V}} = \varnothing$$

In other words, we say that a type s is a subtype of a type t, noted $s \leq t$, if and only if the set of values denoted by s is contained (in the set-theoretic sense) in the set of values denoted by t. A variation of the Castagna type system, as presented in [19], extends asynchronous π-calculus [16] with semantic characterization of channel types supporting semantic subtyping and type inference.

4.2.3 Weaknesses in existing used type systems

The lack of support of type systems for service-oriented systems has been cited in [66, 71]. The existing type systems for services are of limited expressiveness compared to type systems that have been proposed for other software systems.

Moreover, support for service subtyping is also lacking, as has been recognized recently in several different contexts [50, 47]. In [47], the authors discusses the need for sophisticated service discovery, in particular supported by a sound type system with subtyping. In a related context, testing for subschema relationships of XML Schemas is known as strong requirement for service interoperability [50].

4.2.4 Type safety and type checking

Most of existing type systems are not protected against attackers (with the notable exception of that by Hennessy and Riely [68]). An agent may discover a new service then has to adapt its local typing context accordingly. Furthermore in case of multiple discoveries of a channel, the agent has to refine the type of that channel. But malicious agents may change the type of values, including channel values in our system, while they are sent as part of messages via the network. It is therefore useful to consider how types can be protected in the presence of malicious agents as well as insecure channels.

Moreover, guaranteeing the correctness of service applications in a highly-dynamic contexts is an important open issue, as well in practice as in research. Type checking of methods and services is a well-known and proven means to (statically or dynamically) enforce properties, such as security properties and program transformations for the optimization of service composition. Type checking is also fundamental for the discovery process in SOA. In a dynamic context, services may be modified and new services be discovered at runtime. Thus checking if a provided service is conform to the required one becomes a fundamental need. Already existing technologies for discovering Web services, like UDDI and WS-Discovery, lack the sub-typing mechanism in their discovery process in order to enable required services to be provided by more specific ones.

There are few references discussing type checking in the context of Web services. Pu in [66] defines a type system for semi-structured data with application to a wide range of data models and query systems: relational data bases, XML documents, Web services, etc. The type system is based on a nested record type system with collection and universal polymorphism. The author shows that it can easily integrate the sum of types. This type system is neither recursive nor it allows channel mobility. The principle of the type checking is unification but basically it is intractable. Pu studies unambiguous unification to get a polynomial time checking. The work

of Sans and Cervesato [71] deals with an abstract model for web applications. It covers code mobility. They present a general view of the Web services interactions but without parallelism and asynchrony. Their type system does not allow recursive types and they use a global service typing table collecting types of services published everywhere in the Internet. Moreover, they assume trust of the typing information contains in the global repository. Thus they provide a language and a prototype allowing mobile code and remote code. While they allow channel mobility they do not consider channel type discovery and trust of the typing information. A distributed and typed π-calculus for mobile agents is described in [68]. The type system considers malicious agents with erroneous types. Type safety is enforced by dynamically type checking agents when they enter a site. However, they do not consider channel discovery or subtyping. Finally, none of the above approaches discussed flexible typing in the presence of malicious agents and insecure channels.

4.3 Discussion

In order to model the communication between Web services interfaces, the chemical model defined for Join-calculus [28] fits very well. Moreover, the type system defined by Castagna [29] responds to our needs for dynamic discovery and subtyping. However, these two work are more general than what we expect to specify for Web services modeling. Indeed, as the Join-calculus is proven equivalent to the π-calculus of Milner, Parrow and Walker [58] by Fournet in [28], thus it allows to model local orchestrations in addition to the communication level, which is unnecessary for a black box model. Furthermore, the type system of Castagna is very general. Therefore, in order to have a simple model more dedicated to Web services interface-based interaction, it is better to define a model combining the communication part of the Join-calculus and using Castagna type system for dynamic discovery with type inference.

In addition, there is a need to define the safety of such a model by applying type checking in order to avoid errors and by enforcing a security specification which considers the presence of malicious agents in the network able to modify messages content or cause disturbance in the system.

Towards a Well-Founded Object-Oriented Framework for Web Services

5

Inconsistencies in Object-Oriented Frameworks for Web Services

Contents

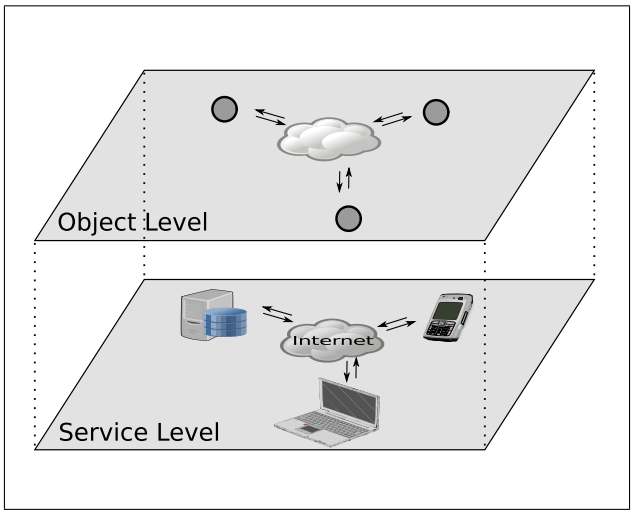


Figure 5.1: Two component levels for Web services communication

In Chapter 3, we presented the OO frameworks for Web services as a composition of two levels: an object level built over a service level as represented in Figure 5.1. From Chapter 2 and Chapter 3, we deduced that there is a need to improve these frameworks in order to allow a unification of dynamic discovery with subtyping and to allow an interoperability by respecting OO subtyping between required and provided services.

In this chapter, we go more in details in the requirements between the two levels in such frameworks. In Section 5.1 We present these requirements following two axes: *i)* `Loose coupling`: allowing the complex technical details of the service level to be hidden at the object level and the service level to be evolved with a minimal impact on the object level, *ii)* `Substitution principle`: allowing the interoperability between Web services by applying OO subtyping. Then, in Section 5.2, we show by examples applied on some existing frameworks (mainly on the `cxf` framework) how these requirements are not fully satisfied.

5.1 Requirements in object-oriented frameworks for Web services

In the following, we present `Loose Coupling` and `Substitution` requirements in OO frameworks for Web services. The two requirements become not only desirable but also expected between the object and the service levels in an OO Web services framework. For each requirement, we present the utilities at service development and execution (if they both exist).

5.1.1 Loose coupling

A `loose coupling` between the service level and the object level would allow evolutions of the service level with a minimal impact on the object level. Such property is interesting since there is a need to deploy the same object model (with minimal adaptations) using the two competing Web services technologies, the `SOAP` and `RESTful`. Requiring a loose coupling between two hierarchical levels 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.

This loose coupling principle makes sense at service development in two forms:

- Dynamic service discovery: a Web service client could use a particular Web service structural interface (`WSDL` or `WADL`) without specifying at development time the related access point service. The developed OO code is thus based on a Web service object type (class or interface) generated from the structural interface by schema compilation. The decision of which service to bind is reported until runtime by querying a discovery registry. In respect with the loose coupling principle, the discovery querying developed at the client object level must be independent from technical details of the discovery model used at the service level. The discovery model can change depending from the service model, `SOAP` or `RESTful`.

- Binding schema: the mapping information used to bind the object types (classes) to structural types (`XML` schema, `JSON` schema) must respect the loose coupling property between the two levels. These mappings must be more adapted to the OO world independently from the structural type details.

5.1.2 Substitution principle

An interoperability induced by the `substitution principle` [51] would allow different substitutions to take place in the object level: a value of a subtype could be exchanged between the client and the server where a value of a supertype is expected. The substitution principle associated to subtyping is of common use in OO programming languages, improving flexibility: implementations can be freely refined whereas clients can be defined from black boxes.

This substitution principle makes sense both at service development and service execution in several forms as we detail in the following.

Service development issue: interface refinement. By applying the `substitution principle` [51], we can define the following `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. Conformally to this rule, a Web service component (client or server) can refine its object types into subtypes. A refined interface is an interface with extra methods or with specialized methods which consume supertypes and produce subtypes, following the well-known variance rules [18].

Service execution issue: dynamic discovery. Service-oriented applications are frequently used in highly dynamic contexts: service compositions may be modified and new services be discovered at runtime. In accordance with the substitution principle previously presented, the interface and the implementation class on the server could be replaced dynamically with a refined interface and its implementation. Thus, the new discovered service could define an interface subtype of the replaced one. Switching from a service interface to a subtype one must be done automatically without any modification at the client side.

5.2 Weaknesses in existing frameworks

Following the existing techniques for Web services development and execution, the previous requirements are not respected. In what follows, we explain three main problems: *i)* tight coupling between the discovery querying at the object level and the used discovery model at the service level, *ii)* weak interoperability with respect to the substitution principle and flexibility between the object level and the service level, *iii)* tight coupling in binding schema.

5.2.1 Tight coupling in object-oriented discovery APIs

The development of dynamic discovery querying at the object level tightly depends from the technical details of the discovery model from the service level. In Chapter 2, we presented several techniques to discover Web services in `SOAP` and `RESTful` models. In the following, we revisit these techniques to show how the service discovery at the object level requires understanding the details of the discovery procedure for each technique. We note here that the followed treated discovery examples concern only the case of a dynamic context where the client proxy is statically defined at development time while the decision of which service to bind is delayed until runtime (see `Updating service access methodology` in Section 2.1.2 of Chapter 2).

`SOAP` services. In the case of a `SOAP` service model, there are two techniques for service discovery:

- `UDDI` standard: `UDDI` defines an inquiry API (a `SOAP`-based interface for querying a `UDDI` server). There are OO APIs that map directly to the `UDDI` Inquiry API (`IBM` and `Systinet` provide one for `Java`). The principle of the discovery process here is to query the `UDDI` registry server about services that implement the `WSDL` associated with a particular `tModelKey`. A `tModel`, or also the `UDDI` technical model, provides a structure to represent a `WSDL` file [8].

  First, the client must define a `UddiRegistry` class. When instantiated, this class creates a reference to a `UDDI` server. It contains one method, `lookup`, that queries `UDDI` for a service that implement the `WSDL` associated with the `tModelKey` and returns a local `proxy` of that service. The `lookup` method takes two arguments: the `tModelKey` of the `WSDL` interface in question, and a Class object that is the service object type (here it is a `proxy interface`). The return value is given as

an instance of the object type specified in the method's second parameter (this is the service proxy). The `lookup` method looks for an access point of a Web service implementing the `WSDL` document linked to the proxy object interface. This is possible by getting a `bindingTemplate` of the `tModelKey`. For the implementation details about the `lookup` method, please refer to Appendix B. This Appendix presents a possible implementation of the `lookup` method using `Systinet` API. This implementation shows how complex the code is. It is required to understand complex details of the `UDDI` standard in order to simply discover an access point service for a given object interface.

Second, in order to show how the client can use the `UddiRegistry` class to get a service (a client proxy instance) and to call it, we take an example of a flight reservation service. The client, knowing a `WSDL` interface of such a service but not its access point, defines a required `Java` interface `FlightReservationInterface` to call a `bookTravel` operation. The client get a `flightReservationService` instance, result of calling the `lookup()` method using a corresponding `tModelKey` and the Class object `FlightReservationInterface`.**class**. The following listing shows more details about the implemented code.

```
public static void main(String args[]) throws Exception
    {
    // Well known tModelKey of the WSDL
    TModelKey tModelKey =
            new TModelKey("uuid:c01dd3c0-f83e-11d7-bbaa
                -b8a03c50a862");
    // Instantiating a new UDDIRegistry bound to the
       inquiry port of a distant
    //  UDDI server
    UDDIRegistry uddiRegistry =
                new UDDIRegistry("http://uddi/inquiry
                    ");
    // Finding a service proxy for one of the possibly
       many
    // services that implement the
       FlightReservationService interface
    FlightReservationInterface proxy =
      (FlightReservationInterface) uddiRegistry.lookup(
         tModelKey,
      FlightReservationInterface.class());
    // Asking the proxy to book a travel and getting
       back the reply from the server
    BookingReply  reply = proxy.bookTravel(new
       BookingRequest("Travel002-Paris-Berlin"));
   }
```

The `FlightReservationInterface` and its associated classes are defined as

follows:

```
public interface FlightReservationInterface{
public List<TravelReply> searchTravel(TravelRequest treq);
public BookingReply bookTravel(BookingRequest breq)
}

public class TravelRequest{
  private String source;
  private String destination;
  private String date;
  // Getters and Setters
}

public class travelReply{
  private String id;
  // Getters and Setters
}

public class BookingRequest{
  private String travelId;
  // Getters and Setters
}

public class BookingReply{
  Boolean confirmation;
  // Getters and Setters
}
```

- `WS-Discovery` standard: In such a standard, the discovery search is available through a `probe` operation. This operation requires a `QName` argument and returns a set of `EndPoint References`.

  The `cxf` framework[1] provides an API to `probe` a `WS-Discovery` proxy. The `org.apache.cxf.ws.discovery.WSDiscoveryClient` class[2] provides several `probe` operations for probing the network. We take for example the following `probe` operation:

  `List<EndpointReference> probe(QName type)`

  The argument of the `probe` operation is a class `QName`[3] which corresponds to the `QName` argument of the `probe` operation as defined in the `WS-Discovery` specification. In the same manner, the `List<EndpointReference>` is the `Java` representation of the `EndPoint References` type. In the following listing, we show

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

[2] http://grepcode.com/file/repo1.maven.org/maven2/org.apache.cxf.services.ws-discovery/cxf-services-ws-discovery-api/2.7.3/org/apache/cxf/ws/discovery/WSDiscoveryClient.java#WSDiscoveryClient.probe%28%29

[3] http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/javax/xml/namespace/QName.java#QName

an example of service querying using the cxf API for WS-Discovery. As for the UDDI previous example, we consider here a flight reservation service associated to a FlightReservationInterface interface.

```
public static void main(String args[]) throws Exception {
  // Using WS-Discovery to find references to services that
      implement a flight reservation service interface
  WSDiscoveryClient registryClient = new WSDiscoveryClient
      ();
  List<EndpointReference> references = registryClient.probe
      (new QName("http://FlightReservation/", "
      FlightReservationService"));
  registryClient.close();

  // getting a random reference value from the references
      list,
  // we consider the first element in the list
  EndpointReference ref = references.get(0);

  // getting the proxy from the reference
  FlightReservationInterface proxy = ref.getPort(
      FlightReservationInterface.class, null);

  // asking the proxy to book a travel and getting back the
      reply from the server
      BookingReply reply = proxy.bookTravel(new
          BookingRequest("Travel002-Paris-Berlin"));

}
```

The FlightReservationInterface is the same as defined previously for the UDDI case.

The probe method is already defined in the cxf WS-Discovery API contrary to the lookup() method (the equivalent of probe method) for the UDDI case where the client has to define it. Indeed, each OO API, for WS-Discovery or UDDI standards, maps directly the defined methods in these standards. For the UDDI case, the lookup() method is the result of calling multiple UDDI standard methods.

RESTful. Let us take again the previous discovery scenario presented for the SOAP case. We consider here that a client would like to discover dynamically a flight reservation service using links between resources. To simplify, we consider that the client directly query the root-resource of the flight reservation service. Discovering the flight reservation service means discovering two sub-resources: travel resource and booking resource. Searching for a travel corresponds to calling the GET method on the travel resource. Booking a travel corresponds to calling the PUT method on the booking resource. Next,

we present the code to discover the flight reservation service using the resource link discovery with RESTEasy and Atom links[4]. First, we show in the following listing the code at the server side. We consider again the FlightReservationInterface previously defined. We define a RootFlightReservationResource interface and a FlightServiceDescription. The RootFlightReservationResource interface defines one provided method, getSubResources in order to return the wanted descriptions of the two sub-resources (an instance of FlightServiceDescription class). The FlightServiceDescription class is composed from two attributes: TravelResourceDescription and BookingResourceDescription.

```
// Root resource interface
@Path("/FlightReservationService/")
public interface RootFlightReservationResource{
@AddLinks
@Get
public FlightServiceDescription getSubResources();
}

// Flight reservation service interface
@Path("/FlightReservationService/")
public interface FlightReservationInterface{
@LinkResource(value=TravelResourceDescription)
@Get
@Path("travel/")
  public List<TravelReply> searchTravel(TravelRequest treq);

@LinkResource(value=BookingResourceDescription)
@PUT
@Path("booking/")
  public BookingReply bookTravel(BookingRequest breq);
}


@XmlRootElement
public class FlightServiceDescription{
  private TravelResourceDescription travelResource;
  private BookingResourceDescription bookingResource;
}

@Mapped(namespaceMap=@XmlNsMap(jsonName =  "atom", namespace = "http
    ://www.w3.org/2005/Atom"))
@XmlRootElement
public class TravelResourceDescription{
  RESTServiceDiscovery  atom;
}
```

[4] http://www.google.fr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=8&sqi=2&ved=0CGcQFjAH&url=http%3A%2F%2Fa3.mndcdn.com%2Fimage%2Fupload%2Ft_attachment%2Fx8cjgapgcfh7wilnvtzy.pdf&ei=O533UoupMIyX0AWD_4CACA&usg=AFQjCNGl9OzXQVYpBbifhY19LXsF65-Spw

```
@Mapped(namespaceMap=@XmlNsMap(jsonName =  "atom", namespace = "http
    ://www.w3.org/2005/Atom"))
@XmlRootElement
public class BookingResourceDescription{
  RESTServiceDiscovery atom;
}
```

The annotations used in the previous Java code such as:

- @XmlRootElement is specific to JAXB data binding
- @Path, @Get, @Put are specific to the JAX-RS API which allows to specify the URI path to access a CRUD method.

To define the link to a flight reservation service, three things are used in order to tell RESTEasy to inject Atom links:

- @AddLinks annotation on the RootFlightReservationResource method (getSubResources) to indicate that Atom links must be injected in the response entity,
- RESTServiceDiscovery[5] field added to each sub-resource description class, TravelResourceDescription and BookingResourceDescription where Atom links must be injected,
- @LinkResource annotation on the FlightReservationService class methods so that RESTEasy knows which links to create for the service methods: the searchTravel method is linked to the TravelResourceDescription and bookingTravel method is linked to the BookingResourceDescription.

At the client side, the same interfaces and classes as at the server are defined, except that the FlightReservationService required interface does not specify of course the paths URI on searchTravel and bookTravel methods. If the client calls the getSubResources() method on its required interface RootFlightReservationResource, it will then get the following XML representation:

```
<FlightServiceDescription>
  <TravelResourceDescription>
    <atom:link href="http://FlightReservationService/travel/" rel="
       self"/>
  </TravelResourceDescription>
  <BookingResourceDescription>
    <atom:link href="http://FlightReservationService/booking/" rel="
       add"/>
  </BookingResourceDescription>
</FlightServiceDescription>
```

[5] https://docs.jboss.org/resteasy/2.0.0.GA/javadocs/org/jboss/resteasy/links/RESTServiceDiscovery.html

The client discovery query code to book a flight may look like the following:

```
public static void main(String args[]) throws Exception {
RootFlightReservationResource proxy = ProxyFactory.create(
    RootFlightReservationResource.class, "http://
    FlightReservationService/");
FlightResourceDescription resourceDesc = proxy.getSubResources();
RESTServiceDiscovery atom = resourceDesc.getBookingResource.
    getAtom();
AtomLink putLink= atom.getLinkForRel("add");
String putURL= putLink.getHref();
RestEasyClient client = new RestEasyClientBuilder().build();
RestEasyWebTarget target = client.target(putURL);
Flight flight = new Flight();
flight.setDate(new Date("10", "Sep", "2013"));
flight.setSourceCity("Paris");
flight.setTargetCity("Beyrouth");
Entity<Flight> request = new Entity<Flight>(flight, null);
Confirmation return =  target.request().put(request, Confirmation
    .class);
}
```

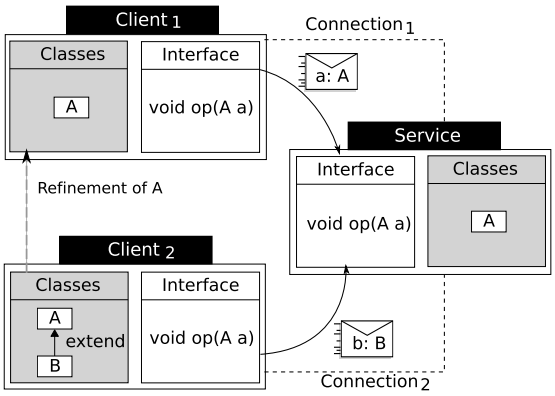
Indeed, booking a flight corresponds to calling the `PUT` method on the flight resource. When the client receives the flight resource description (`resourceDesc` instance), the client get the `Atom` then the URI corresponding to the `PUT` method on the flight resource. Finally, the client calls the `PUT` method with a `Flight` object instance.

Diagnosis. The previous presented `Java` codes in `SOAP` and `RESTful` services, show clearly the complexity of developing a dynamic discovery for querying a service. This complexity is due to the diversity of the existing methods and techniques. That makes the object level tightly dependent from the service level, in some common cases, it is impossible to evolve from one service discovery technology to another, for instance from `UDDI` to `WS-Discovery` in the `SOAP` case, or also from `SOAP` to `RESTful`, without a hard modification of the client developed code.

5.2.2 Weak interoperability in respect with the substitution principle

With an OO 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, (they have the same structural type at the service level, see Definition 1 of Chapter 3 for equivalence). However, from an OO perspective, the tight coupling should not imply usage restrictions since the `substitution principle` [51] can be applied. The question raised in here is the following:

> Is the substitution principle valid in an OO framework for Web services like `cxf`?



(a) Example

	SOAP	RESTful
Interoperability is preserved	**Yes**	No

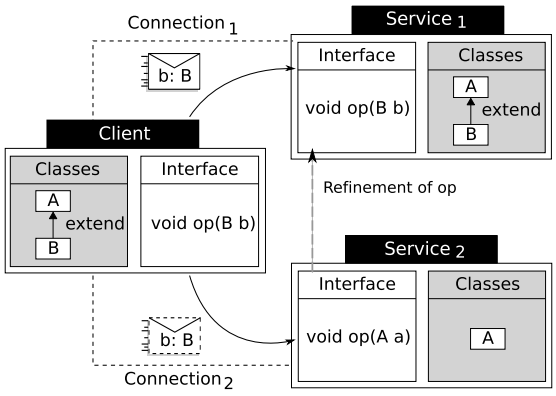
(b) Test results with `cxf` 2.5.10

Figure 5.2: Value substitution

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 5.2(a) represents the interface of a service composed of one operation (method **void** `op(A a)`) and 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 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 proxy 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 5.2(b). 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.

Interface Substitution. Figure 5.3(a) represents two services and a unique client. The client is initially configured to call $\texttt{Service}_1$. $\texttt{Service}_1$ is then replaced with an-



(a) Example

		SOAP	RESTful
Interoperability is preserved	B ≡ A	**Yes**	No
	B ≢ A	No	No

(b) Test results with cxf

Figure 5.3: Interface substitution

other 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 5.3(b), 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 defines an attribute in more than A which has an influence on the corresponding structural types, see Definition 1), 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.

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

- 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.
- Strong coupling: the object and service levels 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.

5.2.3 Tight coupling between binding schema and service technology

As previously defined in Section 3.1.2, binding schemas are used to map object types to their corresponding schema (`XML` schema or `JSON` schema). Actually, these mapping information are not loose coupled with the Web services technology in regards with their definition and use in the OO environment. In the following, we present some disadvantages of the existing binding schema, by considering as an example `JAXB` data binding.

Complexity.

1. Technical syntax: annotations proposed by `JAXB` for binding schema could be used differently in two senses: schema compilation and schema generation. First, for schema generation, the developer has simply to match a class with a document structure. For this end, he must specify a name for the document root element and a name for each attribute in this class. The developer has the choice to select just a set of these attributes to be represented in the structural document. The following listing present an example of a schema generation annotation:

```
@XmlRootElement(name="product")
@XmlAccessorType(XmlAccessType.NONE)
public class Product {

    @XmlElement(required=true)
    protected int id;
    @XmlElement(required=true)
    protected String name;

    protected String description;
    @XmlElement(required=true)
    protected int price;

    public Product() {}

    // Getter and setter methods
    // ...
}
```

The produced XSD schema is represented in the following listing:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/
    XMLSchema">

  <xs:element name="product" type="product"/>

  <xs:complexType name="product">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="price" type="xs:int"/>
    </xs:sequence>
  <xs:complexType>
</xs:schema>
```

Using the annotation `@XmlAccessorType(XmlAccessType.NONE)` on top of the class definition and `@XmlElement(required=`**`true`**`)` on the required class attributes, the developer specifies only the attributes to be represented in the XSD (the `description` attribute, for instance, is not represented in the XSD because it is not preceded by the `@XmlElement(required=`**`true`**`)` annotation). A class attribute could be annotated using `@XmlElement` annotation or `@XmlAttribute` annotation in order to create an XML element or an XML attribute respectively. Multiple other similar annotations exist to annotate in a similar manner a class[6].

Second, for the schema compilation, we start from an existing XML Schema definition and we generate a corresponding annotated class. Contrary to the case of schema generation, annotations refer to the types defined in the XSD file. If we take for example the following XSD file:

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://xml.product"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           xmlns:myco="http://xml.product">

  <xs:element name="product" type="myco:Product"/>

  <xs:complexType name="Product">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="price" type="xs:int"/>
```

[6] http://blog.bdoughan.com/2011/06/using-jaxbs-xmlaccessortype-to.html

```
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The generated source code for Java classes is represented in the following listing:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Product", propOrder = {
    "id",
    "name",
    "description",
    "price"
})
public class Product {
    protected int id;
    protected String name;
    protected String description;
    protected int price;

    // Setter and getter methods
    // ...
}
```

The attributes of the generated Java class, Product, are the same as the elements inside the corresponding complexType, and the class contains getter and setter methods for these fields. The @XmlType annotation here has the same role as the combination of @XmlRootElement and @XmlElement used for the previous schema generation example.

We deduce that all these technical details are useless for a developer who would like to dissociate the object level from the service level.

2. Confusing use: JAXB requires an @XmlRootElement annotation on the main object class (marshallable type), while it is not mandatory for the nested object classes. JAXB is able to consider a default schema binding for the nested object classes.

   We consider an example of a service operation op with argument A:

```
...
public void Op(A a);
....
```

   We would like to publish this method first as a RESTful service then as a SOAP service. In case of a RESTful service, the marshalling object is the operation argument (instance of class A). Thus, it is mandatory that class A has a root element annotation. While in case of a SOAP service, the marshalling object is a command class, C_{in}, such that:

```
Class C_in{
public A a;
...
}
```

The C_{in} class is often not directly implemented by the developer, but generated when deploying the OO code as a `SOAP` service. In this case, the root element annotation must be located at C_{in} class while the `A` class could be not annotated.

We deduce that the use of annotations in a class depends on the Web services technology.

Subtyping difficulties. A developer may like to define a Web service operation with subtypes. In an OO language, that corresponds to:

- inheritance between classes: Using `JAXB` data binder, the developer has to care about specifying an `@XmlSeeAlso` annotation at the supertype (superclass). Indeed, `JAXB` marshals (unmrshals) objects (structures) into structures (objects) depending from the classes bounded in the `JAXB` context. The `@XmlSeeAlso` annotation allows the `JAXB` runtime to also bind other classes when it binds the referred class. For instance, if we take an example of an operation $op : A \rightarrow void$, which accepts data of type `A` or of type `B` subtype of `A`, the developer has to annotate the `A` class as in the following:

```
@XmlSeeAlso{B.class}
Class A{
...
}

Class B extends A{
...
}
```

- use of interfaces: Similarly to the previous case, an `@XmlSeeAlso` annotation is required at the defined interfaces in order to bind all the implemented classes. Moreover, as `JAXB` needs a `marshallable type` that can be instantiated, while it is not the case for an interface, `JAXB` provides a solution by defining an adapter (`AnyTypeAdapter`)[7].

  For instance, if we take an example of an operation $op : I \rightarrow void$, such that `I` is an interface with two implementing classes, `A` and `B`. The service class must be defined as in the following:

[7] https://jaxb.java.net/guide/Mapping_interfaces.html

```
Class Service{
public void op(@XmlJavaTypeAdapter(AnyTypeAdapter.class) I
    i) {...}
}
```

The `I` interface, `A` and `B` classes must be defined as in the following:

```
@XmlRootElement
@XmlSeeAlso({A.class , B.class})
Interface I {...}

Class A implements I {...}

Class B implements I {...}
```

Multiple disadvantages are present due to such a development practice:

1. It is not natural that the developer care about this subtyping details in an OO context. Handling subclasses should be done by default in such frameworks. Indeed, if a service can treat a super type then it is able to treat a subtype even if only the supertype is known.
2. It becomes more difficult to update the OO code by adding other subtypes. Let us say, a developer would like to add a new class `C`, subclass of `A`, for the previous presented `op` service. He has to update the `@XmlSeeAlso` annotation on the superclass (or interface) in order to add also class `C` to the list as in the following:

   ```
   @XMlSeeAlso{B.class , C.class}
   Class A{
   ...
   }

   Class B extends A {...}
   Class C extends A {...}
   ```

   Things start to be more complex when there are multiple nested classes (interfaces). It would be more adequate if an annotation would be on the new subclass.

Update difficulties. The existing OO frameworks for Web services developments already provides tools for schema generation and schema compilation (see the definition in Section 3.1.2) which help to generate the associated schema mappings. In addition, there is an automatic generation of implementation codes for Web services, clients and some execution tests, like in J2EE and .NET platforms. These benefits help developers to properly create their codes and to avoid type errors. While these frameworks give development aids, some developers begin to rely on these tools without understanding the

code that is automatically generated. However, the schema binding annotations and their assigned code have an important role in type checking messages: they are a part of structural typing for service documents at the service level. Problems can appear when the developer has to intervene to update binding schema: for instance, the developer can wish making some modifications in its implemented service code or simply change the service partner in interaction with. In such case, developer can make mistakes and therefore type checking fails.

In conclusion, schema binding must be standardized such that it becomes more adequate with the OO thinking development and loose coupled with the technical details of the service level.

5.3 Conclusion

In this chapter, we showed the weaknesses in the existing OO frameworks for Web services in respect with `loose coupling` and `substitution principle`. We mainly detailed three problems:

- Tight coupling between the discovery querying at the object level and the used discovery model at the service level: following three examples using `Java` APIs for `UDDI`/`WS-Discovery` standards (for `SOAP` discovery) and `Atom links` (for `RESTful` discovery), we proved that the used discovery protocol at the service level is not transparent for development at the object level.

- Weak interoperability in respect with the substitution principle: following two examples applied on the `cxf` framework for `SOAP` and `RESTful` models and which illustrate the application of the substitution principle (on values and interfaces), we proved that there is a lack of interoperability and a strong coupling between clients and servers when OO subtyping is used.

- Tight coupling in binding schema: following some examples on `JAXB` data binding, we showed that the mapping between object types and their corresponding schema tightly depends from the Web services technology and does not fit with modular OO development practices (subtyping and updates).

Problems presented in this chapter are mainly due to the diversity of existing implementation techniques of Web services at the service level and which has bad effects at the object level. As we already discussed in Chapter 3, these problems can be resolved if these Web services technologies could be unified on common concepts and under a whole abstract behavior.

In the next chapter, we introduce a typed unified model which will constitute the foundation of our work analyses in order to resolve presented problems in this chapter.

6

A Black Box Formal Model for Service-Oriented Computing

Contents

In this chapter, we provide a formalization of a black-box model for service communications that abstracts from implementation details and differences of existing Web services technologies. This model supports message-oriented services in the presence of discovery and subtyping.

In Section 6.1, we present a formalization of Web services components and an operational semantic for messages exchange. In Section 6.2, we present our type system in respect with the

requirements of a Web services environment. Our type system includes communication using typed first-class channels, general set operators over types, and a subtyping relation. Finally, in Section 6.3, we extend our context to include malicious agents. We show the soundness of our type system, even in the presence of attackers and insecure channels. A combination of authentication techniques with typing ensures a secure typing in insecure environments.

6.1 Web services communication

Our model belongs to the class of `message-passing models` [49]. In the following, we precisely describe the formal model, dealing with the communication of messages. The two main requirements of our model, asynchronous communication and true concurrency, has led us to resort to a chemical model [12]. The operational semantics of our model is therefore given by a chemical abstract machine. We start with the syntactic part, which describes components. Then, the deployment of these components produces a set of particles in an aether containing them. Semantics rules then describe how the aether evolves: there are structural rules and reduction rules.

6.1.1 Components definition

Web services Environment	$ws ::= \gamma$	Component
	$\mid \gamma \parallel ws$	Components in Parallel
Component	$\gamma ::= a[\sigma][I]$	Agent a with State σ and Interface I
Agent	$a \in \mathcal{A}$	Set of Agents
State	$\sigma \in \Sigma$	Set of States
Interface	$I ::= \mathsf{0}$	Empty Interface
	$\mid I \,\&\, c^{io}$	Interface Compound with Channel
Channel	$c^{io} ::= c^{in}$	Input Channel
	$\mid c^{out}$	Output Channel
	$c^{in} ::= k^{\iota}$	Input Channel k
	$c^{out} ::= k^{o}$	Output Channel k
	$k \in \mathcal{K}$	Set of Channels

Table 6.1: Components and Agents

We assume that a Web services environment is described as a set of components in parallel.

A component is defined by an agent name a, a state σ and an interface I. The state of components is kept abstract, in accordance with the black-box principle. Different formalisms, like process algebras, could be used to model agents internal behaviors. An interface can be empty (0) or declares (the names of) input and output channels. Input channels k^ι correspond to the channels provided by the agent: input messages are received on these channels. Output channels k^o have two different roles: (i) they correspond to the channels sending messages to the network, (ii) or they could be communicated to another component by putting them in the message content in order to be discovered at receipt. Table 6.1 defines a formal representation of components and agents respectively.

A component may obey restrictions on agent names and input channels in order to be well-formed:

Definition 2 (Component well-formedness) *A component must satisfy two rules to be well-formed:*

1. *Component Identification – given an agent name a, there is at most one component defining a,*
2. *Channel Univocality – given a channel k, there is at most one occurrence of input channel k^ι.*

6.1.2 Components deployment

Messages exchange between distributed components is assumed to be completely asynchronous: in the absence of failure, messages are eventually delivered but no assumption is made about the delivery duration. We consider the buffer used for communication, as a finite multiset of messages, with no bound and no order (the bag type as in the OCL specification). Actually, a message is defined as some content on a channel. The channel determines the unique target of the message. The model deals with communication failures and component failures in a simple way: messages can be lost, that is not delivered, and components can stop. To avoid dangling messages, a type discipline is defined in the next section (Section 6.2). Components executions are also completely asynchronous: components are truly concurrent, with each its own execution time. Exchanging messages is therefore the only way to coordinate components. Initially, coordination is only possible between components that share a channel, for instance between a server providing a channel and its clients, requiring the same channel. Gradually, components discover new channels since messages can contain channels. We abstract away the content of messages, by using a simple structure, which will be refined in the next section, with a type system.

Aether. The Web services environment defined previously will be deployed in a chemical solution which we call "Aether". The particle can decompose into smaller ones, still static. During aether's evolution (by reduction), light mobile particles appear: they correspond to output messages emitted by agents ($a[k^o(v)]$), messages in transit ($k(v)$) and input messages that are to be received by agents ($a[k^\iota(v)]$).

Aether	Ω	$::= \langle\, \overrightarrow{\mu} \,\rangle$	Multiset of Particles μ
Particle	μ	$::= \gamma$	Component
		$\mid a[\sigma]$	Agent with State
		$\mid a[I^{\iota}]$	Agent with an input interface
		$\mid a[I^{o}]$	Agent with an output interface
		$\mid a[m^{io}]$	Local Message m^{io} at Agent a
		$\mid k(v)$	Message in Transit
Local Message	m^{io}	$::= m^{in}$	Input Message
		$\mid m^{out}$	Output Message
	m^{in}	$::= k^{\iota}(v)$	Input Value v on Channel k
	m^{out}	$::= k^{o}(v)$	Output Value v on Channel k

Table 6.2: Aether

As different messages can have exactly the same form (same channel, same content), the aether is defined as a multiset $\langle\, \mu_1, \ldots, \mu_n \,\rangle$ of particles $\mu_1, \ldots, \mu_n$. Table 6.2 formally sums up this description. In this table, we note by I^{ι} the set of input channels defined by a component (the input interface). Similarly, we note by I^{o} the set of output channels known by an agent (the output interface). We assume that $I^{\iota} \subseteq I^{o}$ because input channels could be also transmitted as a content in sent messages: each input channel is also an output channel in the sense that it could be communicated to another component.

Aether represents the global state of the computing world of components. We abstract away message contents and strictly consider constructions required with a black box view of Web services communications. Message values in our model carried by channels support base values, mobile channels, pairs of two values and left and right injections associated to disjoint union of values.

$$\begin{array}{l} p_1 \parallel p_2 \rightleftharpoons p_1\,, p_2 \\ a[\sigma][I] \rightleftharpoons a[\sigma]\,, a[I^{\iota}]\,, a[I^{o}] \end{array}$$

Table 6.3: Aether – Structural Rules

Structural and reaction Rules. The structural rules given in Figure 6.3 essentially describe the decomposition of components in the aether. Applied from left to right as reduction rules they converge to a normal form. The structural rules repeatedly reduce parallel components to particles until they have been reduced to individual agents.

The interfaces of agents are then decomposed in their input and output interfaces. Besides structural rules, we use two standard inference rules that are common to all chemical abstract

$\dfrac{\overrightarrow{l} \longrightarrow \overrightarrow{r}}{\langle \overrightarrow{l[\sigma]} \rangle \longrightarrow \langle \overrightarrow{r[\sigma]} \rangle}$	reaction law
$\dfrac{S \longrightarrow S'}{S \ , \ S'' \longrightarrow S' \ , \ S''}$	chemical law

Table 6.4: Chemical and reaction laws

machines [12], the `reaction law` that allows schemata to be instantiated in an aether, and the `chemical law` that defines how local reactions are fired in an aether as they are presented in Table 6.4. The reaction laws states that rules are in fact schemata which can be instantiated as soon as there are particles in the aether which match the patterns of the rules. The chemical law means that any reaction can be performed freely in any chemical solution.

Reduction Rules. Reduction rules (see Figure 6.1) describe the, typically irreversible, evolution of the aether. They assume that the aether has been transformed by structural rules into normal form. We define a function $\mathrm{K} \ : \ \mathcal{V} \rightarrow 2^{\mathcal{K}}$, while $\mathcal{V}$ is a set of values and $\mathcal{K}$ is a set of channels. This function maps each value v to the set $\mathrm{K}(v)$ of channels occurring in v. This assumption suffices to account for channel mobility. The set of channels is assumed to be infinite, dynamic creation of new channels is allowed. Three reduction rules are needed for a Web services interaction:

- Rule `[LOC]`: Agent a consumes a, possibly empty, multiset of input messages m^{in}, and produces another, possibly empty, multiset of output messages m^{out}, and updates its state from σ_1 to σ_2.
- Rule `[OUT]`: Agent a sends the message $k(v)$ over the network. A local condition must be met: all the channels occurring in the message ($\{k\} \cup \mathrm{K}(v)$) must be output channels (I^o) declared by the agent.
- Rule `[IN]`: Agent a receives message $k(v)$ from the network. A local condition must be met: the message channel k must be declared as an input channel by the agent. Moreover, the agent upgrades its declaration of output channels by adding all the channels discovered in the content v of the message ($\mathrm{K}(v)$).

We could now state a first soundness property. Assume a well-formed component, satisfying the following property, expressing interface consistency: given an output channel k^o declared by an agent, there is a corresponding input channel k^ι declared by some agent. Then, during its execution, there is no dangling message in the aether, that is no message $k(v)$ without an agent declaring k^ι as an input channel. We do not formally prove this property here as we refine it in the next section, with type soundness. It follows from the conditions about well-formedness, the static check about interface consistency and the two dynamic checks in Rules [IN] and [OUT], as well as the channel discovery in Rule [IN] for a message correctness. Indeed, a message is `correct` if and only if all the channels it contains are defined. Therefore, discovered channels at reception are correct references to existing provided channels which preserve the interface consistency property. A first simple result can then be shown in the untyped model: well formedness and interface consistency imply that all messages are correct. This has the consequence that

$$
\begin{array}{lllll}
\text{[LOC]} & a[\sigma_1]\,,\overrightarrow{a[m^{in}]} & \longrightarrow & a[\sigma_2]\,,\overrightarrow{a[m^{out}]} & \\
\text{[OUT]} & a[k^o(v)]\,,a[I^o] & \longrightarrow & k(v)\,,a[I^o] & \\
 & & \{k\}\cup \mathrm{K}(v)\subseteq I^o & & \\
\text{[IN]} & k(v)\,,a[I^\iota]\,,a[I^o] & \longrightarrow & a[k^\iota(v)]\,,a[I^\iota]\,,a[I^o\cup \mathrm{K}(v)] & \\
 & & k\in I^\iota & &
\end{array}
$$

Figure 6.1: Aether: reduction rules

no dangling messages occur during aether reduction.

6.2 Typing message-oriented services

This section briefly describes the type system for values and then the principles to type check messages. This type system is defined according to some expressivity needs in SOA. We show that our type system ensures that all messages can always be received by some agent, *e.g.,* that messages do not get stuck.

6.2.1 Typing values

We introduce in this section a type system for the values carried by services, *i.e.,* for the untyped model previously presented. The values are constructed according to the following rules:

$$v \;::=\; b \mid l[v], v \mid k$$

A value v is either a primitive value b or a labeled term $l[v], v$ (that can be used to construct sequences of values) or a channel k. The syntax for types is as follows:

$$t \;::=\; \bot \mid \top \mid B \mid l[t], t \mid < t > \mid t + t \mid t \wedge t \mid \neg t \mid \mu X.t \mid X$$

Types are built from a base type B (denoting a set of values b), from value constructors ($l[\_], \_$) and a constructor for channels $< \_ >$. They can be combined using set operations ($+, \wedge$ and $\neg$). They can also use recursion: recursive types may be unfolded infinitely many times, but values are finite. Some recursive types are not constructive (for instance $\mu X.X$); we therefore consider, as usual, only guarded types: a constructor $l[\_], \_$ or $< \_ >$ must occur between any binder μX and an occurrence of the variable X.

6.2.2 Application to examples

After defining the syntax of our type system, we aim in the following to revisit some examples presented in the state of the art (see Section 4.2.1).

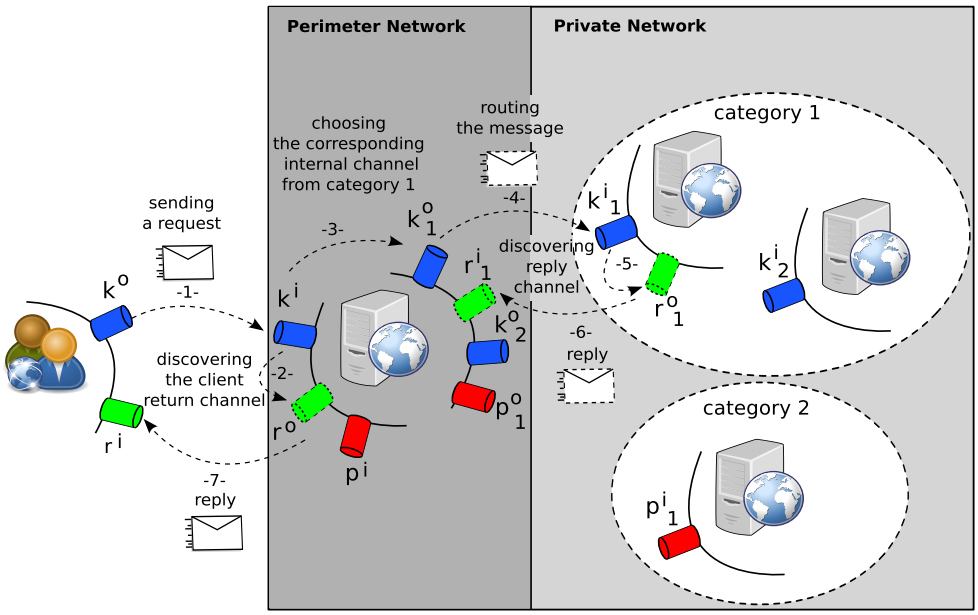


Figure 6.2: Web services routing modeling in our abstract model

6.2.2.1 Channels mobility

We take here the example for Web services routing in order to show how channel mobility could be typed. The formalization of the Web services routing is represented in Figure 6.2. We classify the internal Web services in multiple virtual categories such that channels in the same category have same types. In Figure 6.2, we distinguish two categories: (i) `category 1` which contains two servers having respectively channels k_1 and k_2 such that these two channels are different references to handle a same type of data, (ii) `category 2` which contains a server with channel p_1. For each category, we associate an external channel at the external interface of the perimeter service router: the external channel k is associated to all channels in `category 1` and channel p is associated to the channel in `category 2`. On this external channel, the service router receives a message which contains a data and a channel value the router uses to localize the target internal channel. Indeed, the location of a Web service internal implementation may need to change dynamically due to maintenance processing or to the availability of dependent resources. External clients should be unaffected by these changes. The external channel must have a type that contains the type of the expected data to be routed and a channel type which is the type of an internal channel. At message routing, the service router must infer the type of the return channel of the external client (if it exists in the message content), it is the step 2 in the figure, and associate the target channel to a corresponding internal one, it is the step 3 in the figure. Before sending the message to the internal service (step 4), the perimeter service router must replace the reply channel (if it exists) in the content of the message by specifying another reply channel on its own channels. That ensures the reception of the internal service reply by the router which forwards it then to the external client through its reply channel (steps 5, 6, 7).

The channels of Figure 6.2 example are typed as follows:

Category 1 channels: We suppose that the channels in this category receives a string value and they return an integer value, thus we have:

$$k_1^i : < input[string], < output[int], End > >.$$

This is also the same type for k_2^i channel,

Category 2 channels: We suppose that the p_1^i in this category receives an integer value and returns a boolean value:

$$p_1^i : < input[int], < output[bool], End > >,$$

Perimeter server channels: As k_1^o, k_2^o and p_1^o are the required channels for the corresponding provided ones in categories 1 and 2, thus they have the same type. The reply channel r_1^i has the return type on channel k_1^i thus:

$$r_1^i : < output[int], End >.$$

Channel k^i receives a couple of a string value and a channel, and returns an integer value, thus:

$$k^i : < input[string], < input[string], < output[int], End > >, < output[int], End > >.$$

r^o has the return type of k^i, thus:

$$r^o : < output[int], End >.$$

Finally, we have:

$$p^i : < input[int], < input[int], < output[bool], End > >, < output[bool], End > >.$$

Client channels: the client required channel k^o which has the same type as the corresponding provided channel, k^i. The same is for r^i.

6.2.2.2 Succession of interactions

We take for instance the example of the chain of Web services routing. We consider that the data exchanged in the chain is an integer. The type for the finite chain of channels receiving an integer and a possible channel continuation of the same type is :

$$\mu X.< int > + < value[int], continuation[X], End >.$$

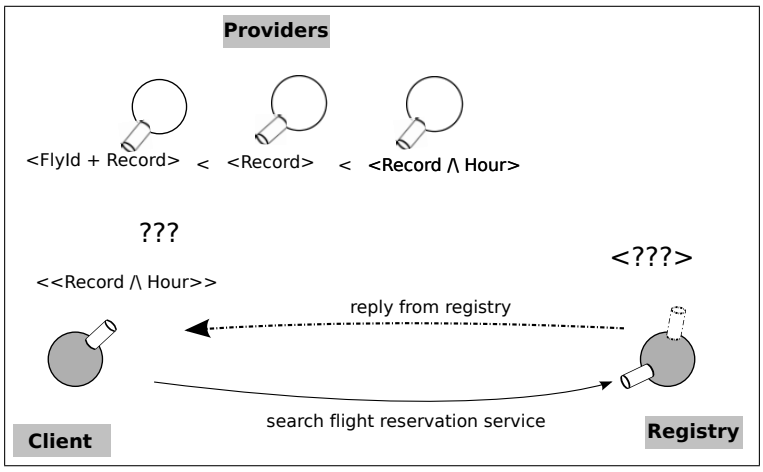


Figure 6.3: Searching a flight reservation service.

6.2.2.3 Customization of Web services interfaces

Let us consider again the flight reservation service, where a reservation is defined using a `Record` type for the departure city, the arrival city and the date of the flight. In order to customize the service interface, the service provider wishes to not handle reservations from and to middle east. For such kind of reservations, we specify a type `MERecord` which is a subtype of `Record`. Thus, the Web service channel type is defined as: **<** $Record \wedge \neg MERecord$ **>**.

6.2.3 Subtyping

Subtyping of services is useful, in particular, in order to type check messages and to enable services to be provided by more specific ones. In the following, we show by an example the utility of subtyping in a Web services context, by revisiting the example of Figure 6.3 presented in the state of the art (see Section 4.2.1.4) then we project the Castagna et al. work [29] to our type system in order to use their defined subtyping algorithm.

6.2.3.1 Subtyping example

In the following, we detail the subtyping example of Figure 6.3 using our type system. The different types in the flight reservation scenario have the following definition in accordance with our type system:

`Flight Reservation Types:`

FlyId = `Int`

Record = $\mu X.(*)[\top], (X + record[dep[String], arr[String], date[Date], End], \top)$

Hour = $\mu X.(*)[\top], (X + hour[h[Int], min[Int], sec[Int], End], \top)$

<FlyId + Record> = <Int +
$\mu X.(*)[\top], (X + record[dep[String], arr[String], date[Date], End], \top)$>

<Record> = <$\mu X.(*)[\top], (X + record[dep[String], arr[String], date[Date], End], \top)$>

<Record $\wedge$ **Hour>** = <
$\mu X.(*)[\top], (X + record[dep[String], arr[String], date[Date], End], \top)$
$\wedge$ $\mu Y.(*)[\top], (Y + Hour[h[Int], min[Int], sec[Int], End], \top)$>

The syntax's complexity of **Record** and **Hour** types is due to the fact of giving these two types the expressivity of accepting a message containing at least an XML tag labeled with **record** and **hour** respectively. In other words, these two types represent a set of labeled tags with at least one of these tags has the label **record** and **hour** respectively. With such a type definition, we can define an intersection type, **Record** $\wedge$ **Hour** which signifies that a message must contain at least two tags with **record** and **hour** labels without a specific order between these two labels. According to this definition, in the following we present a simpler syntax than the previous defined types:

Flight Reservation Types:
_____

FlyId = Int

Record = $record : \{dep[String], arr[String], date[Date], ...\}$

Hour = $hour : \{h[Int], min[Int], sec[Int], ...\}$

<FlyId + Record> = <Int + $record : \{dep[String], arr[String], date[Date], ...\}$>

<Record> = <$record : \{dep[String], arr[String], date[Date], ...\}$>

<Record $\wedge$ **Hour>** =<
$\{record : \{dep[String], arr[String], date[Date]\}, hour : \{h[Int], min[Int], sec[Int]\}, ...\}$>

We consider t, the type of the flight reservation channel:

- If $t =< Record >$:
  - value v_1:

    $$v_1 = record[\ dep[paris], arr[berlin], date[09/07/2012], end\ ], end$$

    is a correct value.
  - value v_2:

    $$v_2 = hour[\ h[13], min[30], sec[0], end\ ], record[\ dep[paris], arr[berlin],$$

$$date[09/07/2012], end\,], end$$

is also a correct value. In this case, the hour tag will be ignored by the type system and only values of the record tag will be taken in consideration.

- if $t =< Record \wedge Hour >$: v_1 is not a correct value because there is not an hour tag while v_2 is admitted.

In the scenario, as described in the state of the art, we considered a query to a registry in order to get a flight reservation service with type $< Record \wedge Hour >$. We admitted that $<< Record >>$ could be a correct output channel of the registry and thus transmitted values could be channels of type either $< Record >$ or $< FlyId + Record >$ according to the contravariance property over channel types. Let us consider now that the registry replies the client by sending a channel K of a server S_1 having the type $< Record >$. At receipt, the client will infer to it the expected type $< Record \wedge Hour >$. That means channel K will be used as a $< Record \wedge Hour >$ typed channel. The messages exchange between server S_1 and the client will match the previous case of study example when we considered the channel $Reservation(< Record >)$ and the value v_2. Thus, a sub-typed value, sent by the client, will be treated as super-typed one at its receipt on channel K at server S_1.

6.2.3.2 Subtyping algorithm

Following Castagna's work [29], our type system could be defined in terms of a semantic type system, that is, in terms of set-theoretic concepts (the operations $+, \wedge$ and $\neg$ are standard set operations). Indeed, the grammar of our type system is nearly the same as in [29]. They defined the following syntax for types:

$$t \ ::= \ \mathbf{0} \mid \mathbf{1} \mid t \rightarrow t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

We can match our syntax with their as:

- $\perp$ matches to **0** and $\top$ to **1**,
- we specify some types as "based" in order to distinguish them from "labeled" types, while they define a general type t without making a particular difference between types,
- the labeled type $l[t], t$ is a restriction of the use of $\times$ operator in the construction of labeled records,
- the channel type, $< t >$ matches with functions type $t \rightarrow t$ because a channel corresponds to a service function,
- the $+, \wedge$ and $\neg$ operators matches with $\vee, \wedge$ and $\neg$ operators,
- for recursive types, there are several ways to formalize them in their general type system: *i)* introduce them with explicit binders $\mu x.t[x]$, or *ii)* define them as regular trees generated by their grammar, or *iii)* define them as the solution of systems of equations. In order to

ensure the definition of a constructive recursive type, they require that every infinite branch has infinitely many occurrences of the $\times$ or the $\rightarrow$ constructors. That matches also with our conditions on the recursive type constructor as defined previously.

Therefore, for our type system, we assume the subtyping algorithm defined in [29].

6.3 Well typed service communication

In this section, we present the fundamental correctness properties of the type system, in particular, type soundness in the context of malicious agents and insecure channels.

6.3.1 Type checking messages

The type of values that are sent as part of service interactions have to be inferred and type checked. Type inference is used at type checking values which allows to infer the types of channels occurring within values. The type checking algorithm filters a type t with a value v as pattern and computes type constraints for the channel values occurring in v. Checking inference will be noted $[\![t;\, v]\!]$ and the context resulting from inference $\Pi(t, v)$. An inferred context is simply a mapping from channel to types and noted Θ in the sequel. For more details about the type inference and the computation of Θ, please refer to the research report [4].

In the following we need a property that relates this inference algorithm and the subtyping test.

Proposition 1 (Monotony of inference w.r.t. subtyping)

$$\forall t, r, v \,.\, t \;\leq\; r \;\wedge\; [\![t;\, v]\!] \;\Longrightarrow\; ([\![r;\, v]\!] \wedge \Pi(t, v) \;\leq\; \Pi(r, v))$$

This lemma states that if t is a subtype of r and if the inference succeeds with t on a value v then it succeeds for r and the subtyping relation also holds for the inferred contexts. Context subtyping is the covariant extension of the subtyping relation on contexts.

We now show type soundness in the presence of benign agents and secure channels. If a message is received that does not contain a channel that is unknown at the receiving site, a type-check performed during emission is sufficient to ensure correct delivery. However, since services may be discovered dynamically, the type of unknown channels must be inferred using a type inference system. As a prerequisite, it is necessary to check that provided and required interfaces are compliant. We ensure compliance using the subtyping algorithm that is part of our type system.

To type check service communications we enrich our model of services of Section 6.1 with type information and declarations: agents are enriched with type declarations for channels and Θ is used to refer to typing context. A sequence of typed output channels of a well-formed component defines a functional relation from channels to types; it can thus be interpreted as a local typing context and noted Θ^o. Similarly, the typed input interface is noted Θ^ι. In order to formalize interface consistency and type soundness, we first define the notions of correct contexts and correct messages.

$$\text{[OUT]} \quad a[k^o(v)]\,,\, a[\Theta^o] \longrightarrow k(v)\,,\, a[\Theta^o] \qquad k^o{:}{<}t{>} \in \Theta^o \wedge [\![\, t;\, v \,]\!] \wedge \Theta^o\,|_{\mathrm{K}(v)} \leq\ \Pi(t,v)$$

$$\text{[IN]} \quad k(v)\,,\, a[\Theta^\iota]\,,\, a[\Theta^o] \longrightarrow a[k^\iota(v)]\,,\, a[\Theta^\iota]\,,\, a[\Theta^o\ \bar{\wedge}\ \Pi(t,v)] \qquad k^\iota{:}{<}t{>} \in \Theta^\iota \wedge [\![\, t;\, v \,]\!]$$

Figure 6.4: Communication with type inference

Definition 3 (Correct context) *A typing context Θ is* `correct` *in aether Ω and noted $correct(\Theta)$ if:*

$\forall\, k \in dom(\Theta)\,.\, \exists\, t, a, t'\,.\, \Theta(k) = t\ \wedge\ a[k^\iota{:}{<}t'{>}] \in \Omega\ \wedge\ (t'\ \leq\ t).$

Informally, a message is correct in aether Ω if an agent exists with a suitably typed input channel and all its carried channels are defined with a compatible type.

Definition 4 (Correct message) *A message $k(v)$ is* `correct` *if:*
$\exists\, a, t\,.\, a[k^\iota{:}{<}t{>}] \in \Omega\ \wedge\ [\![\, t;\, v \,]\!] \wedge\ correct(\Pi(t,v))$.

We are interested in systems in which only correct messages are sent. To this end, we use a notion of interface consistency which ensures that agent interfaces are appropriately typed. Interface consistency states that all messages sent on an output channel are correct. The interface consistency property is a condition similar to type-based interface compatibility for software components [72]. The principle is that the type of an output channel must be a supertype of its declared input channel (due to the contravariance of channels, this means that values sent over the channel must have subtypes of their declared types).

Definition 5 (Interface Consistency) $\forall\, a, \Theta^o\,.\, a[\Theta^o] \in \Omega\ \Longrightarrow\ correct(\Theta^o)$.

Interface consistency is an assumption which must be checked only at creation time of the aether. In order to avoid illegal messages during runtime, we add dynamic type checks to the original aether reduction rules of Figure 6.1. Two dynamic checks are added to the communication rules [OUT] and [IN]. They ensure that the emitted value and the received value are well-typed. Moreover, they allow the type of the discovered channels to be correctly inferred at the receiving site.

The corresponding modified communication rules [OUT] and [IN] are given in Figure 6.4. Let Θ^o be the typed output interface of an emitter agent, v the emitted value, t the type of information of the outgoing channel. At emission time, the emitter uses its output context to check the type of the emitted value before putting it on the output channel. This check is performed by first inferring $\Pi(t,v)$ and then checking the subtype relation with Θ^o restricted to channels occurring in v. More formally, $[\![\, t;\, v \,]\!] \wedge\ \Theta^o\,|_{\mathrm{K}(v)} \leq\ \Pi(t,v)$. At receipt, the agent will infer a context and add it to its typed output interface. However, it can type new channel as well as existing channel values. Thus we add the condition $[\![\, t;\, v \,]\!]$ and the new output interface is $\Theta^o\ \bar{\wedge}\ \Pi(t,v)$, where $\bar{\wedge}$ is a merging "and" operator for contexts. Merging two contexts consists in the union of channel declarations with an intersection ($\wedge$) on channel types occurring in both contexts.

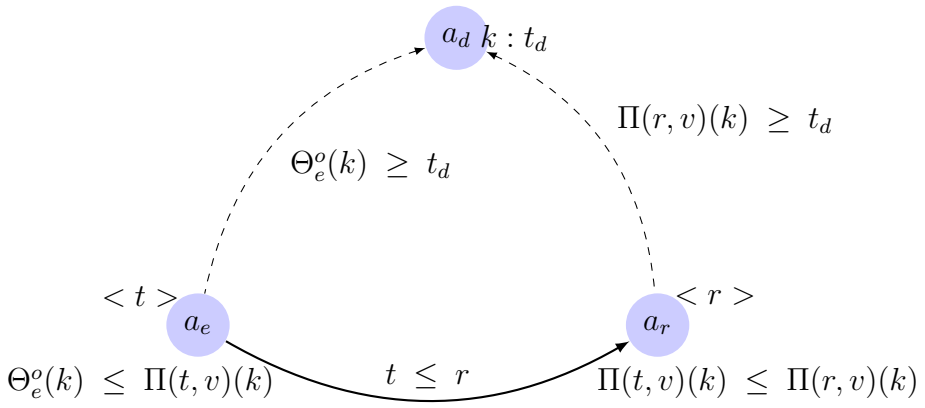


Figure 6.5: Interface consistency preservation

These communication rules with typing conditions ensure that interface consistency is invariant in the presence of channel discovery. For instance, an agent a can create a new channel $k_a :< t_a >$ (provided it is unique). Then this channel value could be sent in a message on a channel k_r to another agent r which is known by a. The discovery mechanism will add to the local context of r the channel k_a with a type which is a supertype of $< t_a >$, thus ensuring interface consistency.

Proposition 2 (Interface consistency invariant) *Interface consistency is preserved by channel discovery at receipt.*

Proof. Figure 6.5 describes the general case related to the interface consistency preservation. Consider that a_e emits a well typed message with value v which contains a channel k ($k \in \mathrm{K}(v)$) known or unknown by the receiver agent a_r. At receipt by a_r we assume that inference succeeds else no discovery is done.

At emission, the type of k is provided by the local context Θ^o_e of the emitter agent a_e. The interface consistency property states that this channel was defined by a, possibly different, agent a_d as a channel with type $\Theta^o_e(k)\ \geq\ t_d$. Let t be the type of this message and $\Pi(t,v)(k)$ the inferred type of the transmitted channel. From the checking conditions in rule [OUT] we have the constraint $\Pi(t,v)(k)\ \geq\ \Theta^o_e(k)$. The type of the message received by a_r is r, but interface consistency and channel contravariance imply that $t\ \leq\ r$. At reception time at a_r, the type inference mechanism initially tags the transmitted channel with, yet another, type $\Pi(r,v)(k)$. By the monotony property of our type inference algorithm, see Proposition 1, $\Pi(r,v)(k)\ \geq\ \Pi(t,v)(k)$ follows. This channel may be new to a_r or it may be already known ($k \in dom(\Theta^o_r)$ for agent r); in the latter case $t_k\ \geq\ t_d$ holds because of interface consistency. The channel k was defined by a_d and we have either $\Pi(r,v)(k)\ \geq\ t_d$ or $(\Pi(r,v)(k)\ \wedge\ t_k)\ \geq\ t_d$ thus interface consistency is satisfied for the channel k for agent a_r. □ ■

Our system enjoys a type soundness property ensuring that every message in transit is correct, which, in turn, entails progress and preservation of message-oriented service communications.

Theorem 1 (Soundness) *If components are well-formed and interfaces consistent, the following property holds:* $\forall\, k,\ v\,.\, k(v) \in \Omega \implies k(v)$ *is correct.*

Proof. The proof relies on two invariants of the aether: component well-formedness and interface consistency. The proof that components remain well-formed is trivial and preservation of the second invariant follows from Proposition 2. Consider a message $k(v)$ that is in transit: it has been inserted in the aether using the [OUT] rule. There is a channel with type $k^o{:}{<}t_e{>}$ and the message was thus type checked within the context of the emitter, *i.e.,* $[\![\, t_e;\ v \,]\!]$. Due to the interface consistency invariant, there is an input channel $a_r[k^\iota{:}{<}t_r{>}]$ and contravariance of channel types implies $t_e \leq\ t_r$, thus we have $[\![\, t_r;\ v \,]\!]$ from Proposition 1. Furthermore, the condition at emitter states that $\Theta^o\ |_{\mathrm{K}(v)} \leq \Pi(t_e, v)$ thus interface consistency implies that the carried channels are defined and well typed. The message is thus correct according to Definition 4. □ ■

6.3.2 Type-checking in the presence of attackers

In this section, we consider networks in which agents may engage in malicious activity or use insecure channels while other parts of the network are composed of trustworthy agents. In this context, we define a message authentication scheme and show how messages can be protected so that malicious agents cannot tamper with message types.

6.3.2.1 Typing error example

Figure 6.6 shows a scenario that illustrates how typing problems may entail security issues. We consider an agent, say `Agent 1`, which probes into a service providing a channel of type $< t >$. Two agents, say `Agent 0` and a `Hacker`, listening on the probing channel, D of `Agent 1`, reply the call. `Agent 0` replies by sending a well typed channel B as the message content while the hacker replies by sending an ill-typed channel K (K has a type $< s > \neq < t >$). Unfortunately, the hacker was faster than `Agent 0`. At message reception on `Agent 1`, channel K is inferred into $< t >$ type. Then, `Agent 1` notifies its neighbors about the new discovered channel: channel K is thus discovered by `Agent 2`, `Agent 3` and `Agent 4`. Consequently, this set of agents will try to reach the channel K by sending ill-typed messages. That can flood the server by erroneous messages and therefore it breaks down. Moreover, the processes in execution at the different agents may waste time due to the generated error. Resolving such a problem may cost in time and in the system performance until identifying the cause of the problem and fixing it again.

6.3.2.2 A need for a weak authentication

In order to avoid such kind of problems, we consider an extension of our aether that may contain uncontrolled agents and monitors (monitors are always trusted). We strive for a provably correct authentication mechanism for systems involving uncontrolled and controlled agents. We

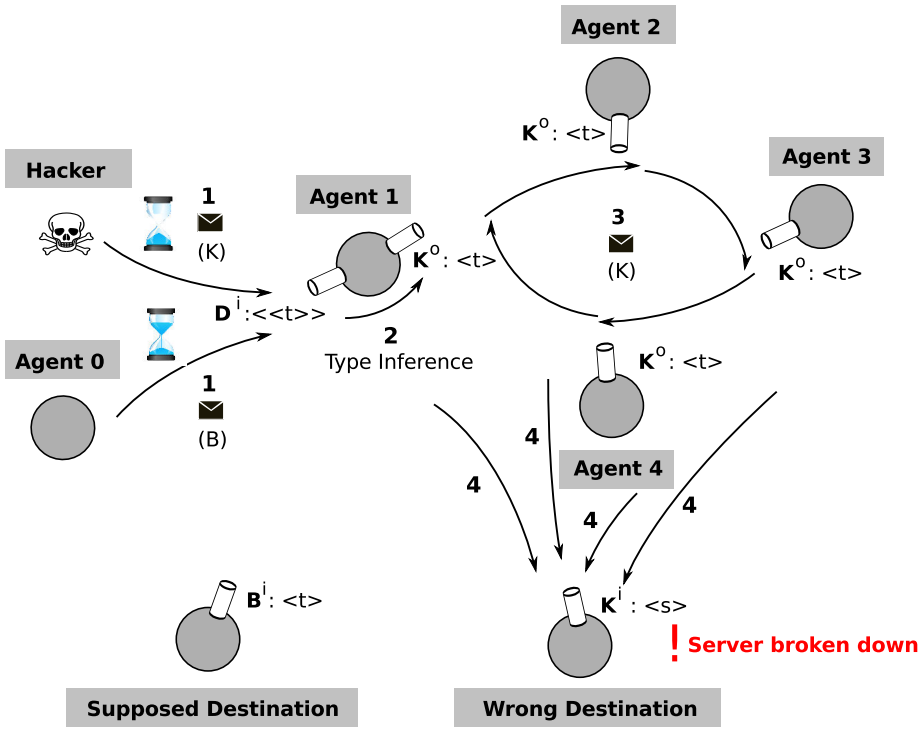


Figure 6.6: typing problems entail security issues

consider a weak message authentication mechanism. More precisely, we support data origin authentication [57] that guarantees integrity of the origin of messages and their uniqueness. That only enables the status (monitored or not) of each message emitter to be ascertained.

In general, messages may be of very different forms because various pieces of information may be added for authentication purposes and uncontrolled agents can forge any kind of message. We restrict the messages to the following two forms: *i)* outgoing messages: $k(v, n, sv)$, and *ii)* incoming messages: $k(v, b)$, where v is a value, b a boolean indicating the status of the emitter (monitored or not), n a nonce and sv a tag (a sticky value that contains, *e.g.,* a secret shared between monitors). A monitor strictly respects the message format but an uncontrolled agent may not. We assume that interfaces of agents are extended to cope with the extra data. We distinguish two kinds of weak authentication:

- authentication through a secure channel: Messages transferred in such channels could not be modified. In this case nonces are not needed to preserve the integrity of the message. Appendix C.1 explains more details about the behavior of a monitor when sending messages to, or receiving messages from, other monitors or uncontrolled agents. In this appendix, we show the modifications required on the reduction rules presented in Figure 6.4.

- authentication through an insecure channel: An insecure channel between two monitors is equivalent to the transmission of messages via uncontrolled agents which act as malicious routers. The challenge is to force the uncontrolled agents to act with integrity by using

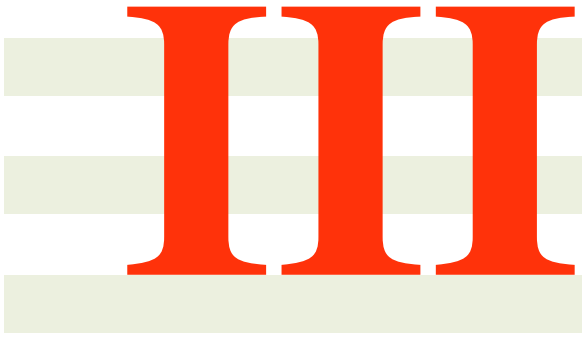
nonces. When a message from a monitor reaches an uncontrolled agent, the latter may suppress it or send another message to monitors or uncontrolled agents. We use hash keys in messages to be able to detect modifications to messages. The behavior of monitors has to be extended so that messages going to or coming from uncontrolled agents are checked for integrity violations. We also need to review the status computation. Appendix C.2 explains more details about this authentication mechanism and how it affects the reduction rules of our model.

Furthermore, the interface consistency property (see Definition 5) for channels needs to be ensured only for the secure (*i.e.,* monitored) part of the network. Uncontrolled agents are not trustworthy; we thus cannot assert properties about their declarations and their behaviors. In Appendix C.3, we discuss again type soundness in presence of attackers.

6.4 Conclusion

In this chapter, we have introduced a high-level model based on a chemical semantics for service interactions and dynamic service discovery with first-class channels. Our proposed model is based on few concepts. Web services are viewed as abstract agents exchanging messages via the network. Services are available thanks to the notion of communication channels. Messages can carry channels, thus ensuring full channel mobility.

This formal model supports a sound type system. The resulting system supports contravariant types for channels, type checking and type inference. We have motivated by examples the utility of our expressive type system for Web services. Furthermore, the type system accommodates subtyping and general set-based type operators. This generality has been achieved by applying the principles of semantic typing to the Web services world. Finally, we have shown fundamental correctness properties of the type system in the context of malicious agents and insecure channels.



A New Specification for a Well-Founded Object-Oriented Framework for Web Services

7

A Unified Object-Oriented API for Dynamic Web Services Discovery

Contents

Multiple OO APIs for Web services discovery already exist in SOAP and RESTful models as we have showed it in Chapter 5. These APIs are tightly dependent on the technical details at the service level which makes their use very complex. Despite the diversity of implementations, the core of the discovery methodology is indeed based on two main points:

1. the service interface,

2. the discovery protocol and the way the registry is called.

In this Chapter, we present how these two points could be unified for the existing service discovery standards in SOAP and RESTful using our formal model presented in Chapter 6. Once we prove the ability to unify the different service discovery implementations at the service level, we are thus able to use this unification to define a unified OO API independent from the technical details. We show how the concepts of the unified formal model could fit with the OO world and how this API should be built conformally to the OO development practices.

The structure of this Chapter is presented as follows. First, in Section 7.1, we present how the standardized Web services components in SOAP and RESTful could be unified using our formal model. Second, in Section 7.2, we show how the different Web services discovery techniques for RESTful and SOAP could be abstracted and how the client/registry interaction could be represented in our formal model. Finally, in Section 7.3, we use the two previous abstractions to define an appropriate OO API.

7.1 Unification of the standardized Web services components

Despite the differences between SOAP and RESTful technologies, we can unify them under the previous presented typed model in Chapter 6. In this section, we show how the different notions of our model match with SOAP and RESTful standards. Moreover, we aim to show how much the existing structural type systems match with our unified type system.

We illustrate our approach through a running example, a service-based system for flight reservation. We consider a two-step scenario: a client component first searches a flight travel from a source city to a target one at a specific date; in a second step, it receives a list of possible flights, makes its choice and books a flight. We investigate two different implementations for this example, respectively using SOAP and RESTful services.

7.1.1 Components and interfaces

In accordance with our model, the flight reservation service component is a composition of an interface and an agent having an internal state which evolves during its execution. The agent abstraction hides all implementation details. The interface is composed of provided and/or required services. Each service is a set of channels to receive incoming messages or to send messages over the network.

First of all, let us formalize the flight reservation scenario in our formal model. The flight service component provides two channels $k_{searchTravel}$ and $k_{bookTravel}$ as defined in the following:

Flight service component

$\gamma_{service}$ = $a_{service}[\sigma_0][I_{service}]$
$I_{service}$ = $k^{\iota}_{searchTravel}$: < travelRequest, <travelReply> >
& $k^{\iota}_{bookTravel}$: < bookingRequest,
<bookingConfirmation> >
travelRequest = request[source[string],
destination[string],
date[string], End], End
travelReply = reply[μX. (End + id[string], X], End)
bookingRequest = travel[id[string], End], End
bookingConfirmation = confirmation[bool], End

$k^{\iota}_{searchTravel}$ channel receives the travel request containing the source city, the destination city and the date (a data of type $travelRequest$) and a return channel to reply the search request with the set of valid travels (data of type $travelReply$). $k^{\iota}_{bookTravel}$ channel receives a travel Id (a data of type $bookingRequest$) in order to book it and a reply channel to reply the booking request with a confirmation of the travel booking (a data of type $bookingConfirmation$).

Now, let us consider the client component which requires the flight reservation service. The formalization of the client component is represented in the following:

Client component

γ_{client} = $a_{client}[\sigma_0][I_{client}]$
I_{client} = $k^{o}_{searchTravel}$: < travelRequest, <travelReply> > &
$k^{\iota o}_{searchReply}$: <travelReply> &
$k^{o}_{bookTravel}$: < bookingRequest, <bookingReply> > &
$k^{\iota o}_{bookReply}$: <bookingConfirmation>

We are not detailing here the defined types as they are the same as at the service provider interface. The two channels provided by the service are required by the client as expressed using the o annotation. Two other channels are declared in the client interface: $k^{\iota o}_{searchReply}$ that gets the list of travels id and $k^{\iota o}_{bookReply}$ used to obtain a booking confirmation. These two channels have a double annotation ι and o: they are at the same time input channels and output channels in the sense that these channels can get out the agent as a content in a message to be discovered at the destination.

In the following, we aim to match the previous formalization of service and client components with concrete concepts in SOAP and RESTful models. For each model, we show how a Web service interface (WSDL or WADL) could be formalized in our formal model.

- SOAP :

For the flight reservation scenario, we suppose that the port address is L_{server} = "http://flight-travel-service", the binding name is FlightTravelServiceBinding and the service interface defines two operations: searchTravel and bookTravel. The searchTravel operation defines an

input message `travelRequest` and an output message `travelReply`. In a consistent manner, `bookTravel` operation defines an input message `bookingRequest` and an output message `bookingConfirmation`. Suppose that the client is accessible via a location L_{client}, which could be:

- a URI address communicated by the client to the server using a `ReplyTo SOAP` header in case the client/server communication is asynchronous (over two distinguished client/server sessions),
- source IP address and port over `http`, in case that the client/server communication is synchronous (usually over one session of the transport protocol).

According to this brief interface description (for a complete definition of the `WSDL` file, please refer to Appendix D.1), we associate channels with their corresponding formal definitions as follows:

- $k_{searchTravel}$ is L_{server}.FlightTravelServiceBinding.searchTravel.searchRequest
- $k_{bookTravel}$ is L_{server}.FlightTravelServiceBinding.bookTravel.bookingRequest
- $k^{\iota o}_{searchReply}$ is L_{client}.FlightTravelServiceBinding.searchTravel.searchReply
- $k^{\iota o}_{bookReply}$ is L_{client}.FlightTravelServiceBinding.bookTravel.bookingReply

Concerning the matching between the formal type and the `WSDL` types (see the Type Definition part of the `WSDL` in Appendix D.1), we have the following correspondences:

- `travelRequest = request[source[string], destination[string], date[string], End], End` corresponds to:

```
<element name="request">
 <complexType>
  <sequence>
   <element name="source" type="string"/>
   <element name="destination" type="string"/>
   <element name="date" type="string"/>
  </sequence>
 </complexType>
</element>
```

- `travelReply = reply[`μ`X. (End + id[string], X], End)` corresponds to:

```
<element name="reply">
 <complexType>
  <sequence>
   <element name="id" type="string" minOccurs = "0"
      maxOccurs = "unbounded"/>
  </sequence>
```

```
  </complexType>
</element>
```

- `bookingRequest = travel[id[string], End], End` corresponds to:

```
<element name="travelId" type = "string"/>
```

- `bookingConfirmation = confirmation[bool], End` corresponds to:

```
  <element name="confirmation" type="boolean"/>
```

- RESTful : The flight reservation service is composed of three resources:

- a root resource which has the root `URL` of the service: L_{root}="http://flightService/"
- a sub-resource of the root, `travel`, accessible by the `URL`: $L1_{server}$ = "http://flightService/travel"
- a sub-resource of the root, `booking`, accessible by the `URL`: $L2_{server}$ = "http://flightService/booking"

The complete definition of the `WADL` file for the flight reservation service is presented in Appendix D.2. The `GET` method of the `travel` resource replaces the `searchTravel` operation of the previous `WSDL` example. The `PUT` method for the `booking` resource replaces the `bookTravel` operation defined in the `WSDL` file.

L_{client} here has the same signification as for the `SOAP` case. We associate the following meaning for our declared formal channels:

- $k_{searchTravel}$ is $L1_{server}$.GET.Request
- $k_{bookTravel}$ is $L2_{server}$.PUT.Request
- $k^{\iota o}_{searchReply}$ is L_{client}.GET.Response
- $k^{\iota o}_{bookReply}$ is L_{client}.PUT.Response

Concerning the matching between the formal type and the `WADL` types (see Appendix D.2), we have the following correspondences:

- `travelRequest = request[source[string], destination[string], date[string], End], End` corresponds to the list of `param` types in the request part of `travel` resource:

```
<request>
 <param type="string" style="query" name="source"/>
 <param type="string" style="query" name="destination"/>
 <param type="string" style="query" name="date"/>
</request>
```

- `travelReply = reply[`μ`X. (End + id[string], X], End)` corresponds to the `reply` type element in the grammars part of the `WADL` file:

```
<element name="reply">
 <complexType>
  <sequence>
   <element name="id" type="string" minOccurs = "0"
       maxOccurs = "unbounded"/>
  /sequence>
 </complexType>
</element>
```

- `bookingRequest = travelId[string], End` corresponds to the `param` type in the request part of `booking` resource:

```
<request>
 <param type="string" style="query" name="travelId"/>
</request>
```

- `bookingConfirmation = confirmation[bool], End` corresponds to the `reply` type element in the grammars part of the `WADL` file::

```
  <element name="confirmation" type="boolean"/>
```

7.1.2 Messages exchange between interfaces

In order to illustrate the role of our model to unify messages exchange for `SOAP` and `RESTful` models, we consider the booking operation of the flight reservation service. The first two rows of Figure 7.1 show message exchanges that are part of the flight booking request. This request may hold a `SOAP` message over a transport protocol (`http`, `FTP`, `SMTP` or others) governed by the `SOAP` standards or a simple `http` message wrapping a payload for `RESTful` services. Our unified model, represented at the bottom row of Figure 7.1, unifies these two message exchange formats because it allows to represent the structure of the message body and it abstracts all the transport header information using channel mobility (for request/reply and discovery mechanisms as it is presented in Chapter 6).

The execution in the aether of the booking step from our scenario is achieved as follows: The client γ_{client} sends a booking request message, *e.g.,* :

$m_1 = k_{bookTravel}(travel(id(\text{"}flight02\text{"})), k_{bookReply})$, which exits from the channel $k^o_{bookTravel}$. The γ_{server} receives the message via the channel $k^\iota_{bookTravel}$. The server discovers the received reply channel and updates its interface by adding the $k_{bookReply}$ as an output channel: $k^o_{bookReply}$: `<confirmation[bool], End>`. Then γ_{server} calls back the client with a confirmation message, *e.g.,* : $m_2 = k_{bookReply}$`(confirmation(true))` which leaves γ_{server} via $k^o_{bookReply}$. For more details about the booking deployment conformally to the reduction rules ([LOC], [IN], [OUT]), please refer to Appendix D.3.

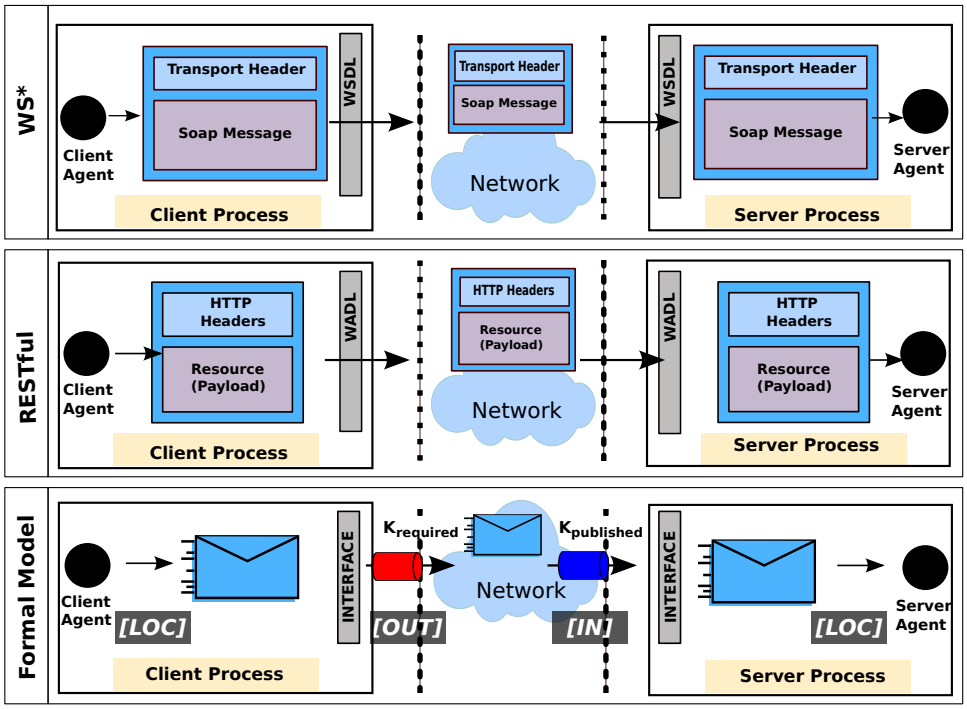


Figure 7.1: Flight booking request: Relationship between SOAP/RESTful and our formal model

Type compliance of exchanged data at [IN], [OUT] rules is ensured through a basic sub-schema compatibility with XML Schema Description (XSD) between the client and the server. Examples and algorithms to process this checking are developed in [50]. In addition to the XML format, in RESTful, data could also be transmitted into JavaScript Object Notation (JSON) format. Some JSON Schemas, much like XSD, exist to verify the structure and data types inside a JSON object.

7.1.3 Matching with the type system

Previously, in Section 7.1.1, we have presented some examples about the use of our type system to type the structure of the flight reservation service. In the following, we revisit the formal type system presented in Section 6.2 to match it with the existing types in the standardized interfaces (WSDL and WADL). We consider here the XSD schema as an example of such types. We show some lacks compared to our formalized expressive type system.

- Labeled type: it is used to represent the tagged values in a structure. It represents a simple type or a sequence in XSD.

- Basic type: it essentially includes the set of well known basic types like: integer, float, string, etc. It could also represent personalized types referring to a set of values.

For instance, if a developer would like to make a type for number upper than 100, he can define a basic type int_2 subtype of int with the desired condition.

- `Channel type`: there is not a type, in clear term, to represent a channel in the existing structural schema. Some cases in `SOAP` and `RESTful` could match with the channel type. First, for asynchronous request/reply case, the client can explicitly specify the return channel. In `RESTful`, it will be an explicit `URL` in the message `http` header. In `SOAP`, it could be a `WS-Addressing`[1] in the `SOAP` message header. Second, for the discovery case in `SOAP` services, there is a `tModel` type defined for the `UDDI` standard to represent a `WSDL` interface (a set of channels in our formal model). Another type could be also compared to channel type in `SOAP`, it is the `EndPointReference` type defined for the `WS-Discovery` standard. For `RESTful`, there is no standard types as in `SOAP`, we can consider for example the `Atom links`, where an `Atom link` is defined.

- `And/Or type constructors`: there is no explicit And/Or type constructors in a schema type. However, in `XSD` for instance, the use of And/Or is implicit in some cases, like for `xs:choice` and `xs:all` constructors. `xs:choice` allows only one of the elements contained in the declaration to be present within the containing element. For instance, if we consider the following `XSD` example:

```
<xs:complexType name="t">
  <xs:choice>
    <xs:element name="x" type="xs:string/>
    <xs:element name="y" type="xs:decimal"/>
  </xs:choice>
</xs:complexType>
```

Type t is represented in our type system as:

$$x[string], End + y[int], End.$$

`xs:all` specifies that the child elements can appear in any order. For instance, if we consider the following `XSD` example:

```
<xs:complexType name="t">
  <xs:all>
    <xs:element name="x" type="xs:string"/>
    <xs:element name="y" type="xs:decimal"/>
  </xs:all>
</xs:complexType>
```

Type t is represented in our type system as:

[1] http://www.w3.org/Submission/ws-addressing/

$$\mu X.(*)[\top], (X + x[string], \top)$$

$$\wedge$$

$$\mu Y.(*)[\top], (Y + y[int], \top)$$

- `Recursive type`: Recursive type appears in schema just to represent lists. It is not possible to represent a recursive channel type.

- `Negation type`: There is not an explicit negation constructor in the existing schemas. However, `XML` Schema 1.1 introduces a new Schema component named <assert> to facilitate rule based validation. <assert> allows to make negation on types because it can specify a Boolean XPath expression.

  For instance, let us consider that a flight reservation service has a type t which specifies the departure city, the arrival city and the date of the flight. The departure city and the arrival city must not be in middle east area. In `XSD`, such type could be represented as in the following:

```
<xs:complexType name= "City" >
 <xs:sequence>
  <xs:element name="name" type= "xs:string" />
  <xs:element name="country" type= "xs:string" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name= "MiddleEastCity" >
 <xs:complexContent>
  <xs:extension base= "City" />
 </xs:complexContent>
 <xs:assert test= "(country=('Lebanon','Chypre', ...) )" />
</xs:complexType>
<xs:complexType name= "t">
 <xs:sequence>
  <xs:element name= "departure" type= "City" />
  <xs:element name= "arrival"  type= "City" />
  <xs:element name= "date" type= "xs:string" />
 </xs:sequence>
 <xs:assert
   test="not (element(departure ,
        MiddleEastCity) or element (arrival, MiddleEastCity)"
          />
</xs:complexType>
```

  Type t is formalized as:

$$departure[City \wedge \neg MiddleEastCity], arrival[City \wedge \neg MiddleEastCity],$$

$$date[string], End$$

$$City = name[string], country[string], End$$

$$MiddleEastCity = name[string], country[substring], End$$

$substring$ is a base type defining the set of country "`string`" values in middle east area.

In conclusion, our type system has direct or indirect correspondences in the `XSD` typing schema. Our type system is richer than the existing structural one particularly in the channel type. However, the existing schema type provides more details particularly for type validation using XPath. Such details are abstracted in our formal model.

7.2 Unification of dynamic Web services discovery protocols

In this section, our goal is to prove that the different existing protocols between a client and a registry for runtime discovery could be unified. For this aim, we use the unified formalization of a structural interface. First, we give an abstract formalization of a dynamic discovery example applied to the flight reservation service. Then, we formalize this example using the different discovery techniques while comparing each one to the abstract formalization. Finally, we deduce a global abstraction which preserves and unifies the concepts of the existing discovery protocols. We show how this abstraction can match with our formal model.

7.2.1 The dynamic discovery mechanism in our formal model

Let us suppose that the client does not know the location of the flight reservation service while knowing its interface. Thus, at runtime, before invoking the service, the client needs to discover a target location for the service. As part of an interface, a service is defined as a set of channels that corresponds to a port in `SOAP` and a set of resources in `RESTful` (as previously presented). Thus service discovery means looking up a set of channels; our model represents such a set in a way compatible with both technologies.

To illustrate the discovery formalization mechanism in the flight reservation service, the client needs to discover the two channels `searchTravel` and `bookTravel` previously described. At design time, the client is implemented with a required interface for such a service: two methods to search a travel and to book a flight. At run-time, to invoke such service, the client asks a registry for a corresponding service. We suppose that there is one component that provides the flight reservation service, it is γ_{server} in our previous components formalization. The registry provides a channel `searchFlightService` for sending the channels of γ_{server}. In the request message on the `searchFlightService` channel, the client sends a reply channel on which it expects receiving two channels:

- $searchTravel$ of type $< searchType >=$ **<** $travelRequest, <travelReply>$ **>**,
- $bookTravel$ of type $< bookType >=$ **<** $bookingRequest, <bookingReply>$ **>**.

The reply channel has thus the type of a couple of two channels:

$$reply :< service[fst[< searchType >], scd[< bookType >], End], End > .$$

In order to simplify the representation of this couple of channels type, we use the following syntactical sugar:

$$reply :<< searchType >, < bookType >>$$

$$reply : \mathbf{<} < travelRequest, \mathbf{<} travelReply \mathbf{>} \mathbf{>}, \mathbf{<} bookingRequest, \mathbf{<} bookingReply \mathbf{>} \mathbf{>} \mathbf{>}.$$

Hence, the formalization of the whole system components in our formal model is defined as in the following (here we show only details concerning the client and the registry components):

%%Global system:
$\gamma_{global} = \gamma_{server} \parallel \gamma_{client} \parallel \gamma_{registry}$

%%Individual Processes:
$\gamma_{client} = a_c[\sigma_{c0}][I_{client}]$
$\gamma_{registry} = a_r[\sigma_{r0}][I_{registry}]$

%%Interfaces description:
I_{client} = searchFlightService^{o}: <DiscoveryMsg>
&reply$^{\iota o}$: < <searchType>, <bookType> >
$=<$
$< travelRequest, < travelReply >>$
,
$< bookingRequest, < bookingReply >>$
$>$

$I_{registry}$ = searchFlightService$^{\iota}$: <DiscoveryMsg>
&searchTravel^{o}: <bookingRequest, <bookingReply> >
&bookTravel^{o}: <travelRequest, <travelReply> >

%%Discovery message type:
DiscoveryMsg $=<$
$< travelRequest, < travelReply >>$
,
$< bookingRequest, < bookingReply >>$
$>$

At execution of the discovery scenario, the client sends a message:

$$m_1 = \texttt{searchFlightService(reply)}$$

to the registry. The registry replies by sending the two channels of the flight reservation service:

$$m_2 = \texttt{reply(searchTravel, bookTravel)}$$

When the client receives the two channels for the flight reservation service, it infers their types and upgrades its interface. This formalization is illustrated in Figure 7.2.

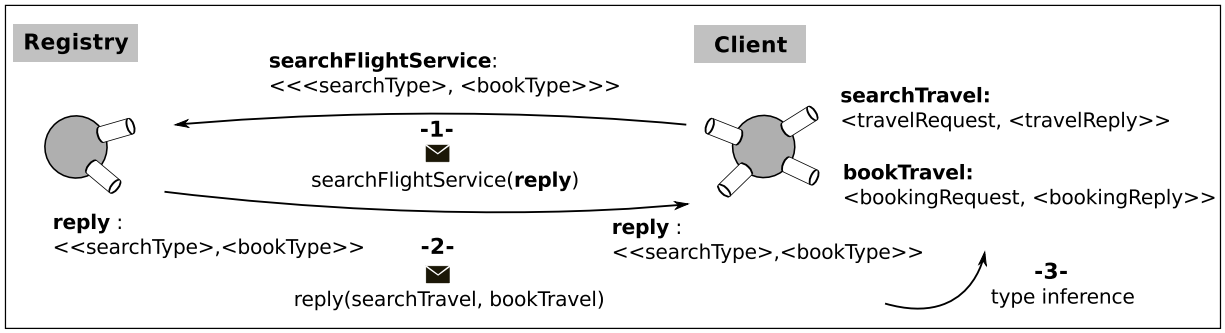


Figure 7.2: Discovery scenario

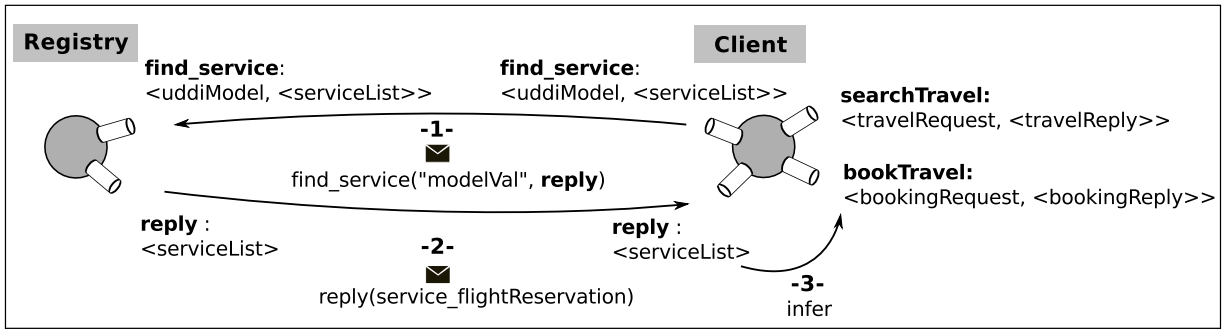


Figure 7.3: Discovery scenario using UDDI standard

7.2.2 A projection to SOAP and RESTful models

In the following, we present how the previous formalized discovery mechanism corresponds to the existing discovery protocols in SOAP and RESTful. We focus here on formalizing the used discovery protocol which differs from one technology to another. As we have proved in Section 7.1.1 that structural interfaces could be unified, the formalization of the required service interface is unchanged for the following treated examples.

- SOAP. In the following, we consider the two discovery standards for SOAP: UDDI and WS-Discovery.

1. For the UDDI standard, the discovery search is available through a $find\_service$ operation defined in the UDDI specification [8]. This operation requires many arguments (we abbreviate them with a $uddiModel$ type) and returns a $serviceList$ type. The return type corresponds to a list of services compatible with the $uddiModel$: services providing the WSDL interface referred by the $uddiModel$. The formalization of this example is represented in Figure 7.3. In order to match this formalization with the abstract one presented in Figure 7.2, the $searchFlightService$ channel corresponds to an operation which calls by default the $find\_service$ operation using a

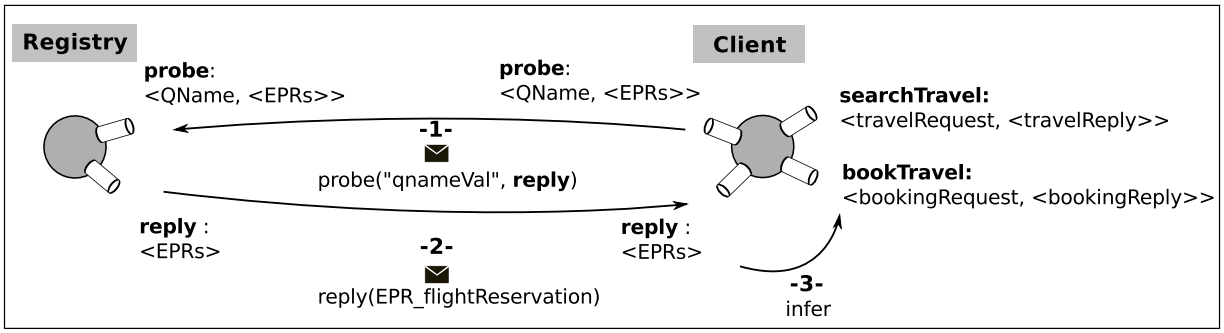


Figure 7.4: Discovery scenario using `WS-Discovery` standard

$uddiModel$ specific to the flight reservation service. This operation should return a selected service from the list of services (of type $serviceList$) returned by the $find\_service$ call. While the type of the reply channel is more generic in the `UDDI` case, it is more precise in the abstraction of Figure 7.2. Therefore, we can deduce the following implementation of the $searchFlightService$ channel for the `UDDI` protocol:

Abstract Implementation 1 (Matching with `UDDI`)

$$searchFlightService : \bot \rightarrow (< searchType >, < bookType >) \Rightarrow$$

$$serviceList\_flightReservation \leftarrow find\_service(uddiModel_{FlightService});$$

$$service\_flightReservation \leftarrow selectElement(serviceList\_flightReservation);$$

$$(searchTravel, bookTravel) \leftarrow infer(service\_flightReservation);$$

$$return \quad (searchTravel, bookTravel)$$

We use the notation: $f : X \rightarrow Y \Rightarrow Z$ to represent a function f having the signature $X \rightarrow Y$ and an abstract implementation Z. Z could be a succession of instructions separated by ";". f corresponds to a channel of type: $< X, < Y >>$. Calling a function f with a set of parameters, p, is represented by: $f(p)$. The result of this call is represented by $R \leftarrow f(p)$.

2. For the `WS-Discovery` standard, the discovery search is available through a $probe$ operation. This operation requires a $QName$ argument and returns a set of $EndPoint - References$ ($EPRs$ type). The $QName$ argument here has the same role as the $uddiModel$ for the `UDDI` protocol: it refers to a particular `WSDL` interface. An $EndPointReference$ is a standardisation specified in the WS-Addressing specification [33] to represent a service. The formalization of such a discovery mechanism is illustrated in Figure 7.4. In the same manner as for the `UDDI` case, the $searchFlightService$ channel of the abstract formalization of Figure 7.2 corresponds to an operation which calls by default the $probe$ operation using a $QName$

specific to the flight reservation service. This operation should return a selected service from the list of $EndPoint - References$ returned by the $probe$ call. The type of the reply channel is also more generic in the `WS-Discovery` case while it is more precise in the abstraction of Figure 7.2. Therefore, we can deduce the following implementation of the $searchFlightService$ channel for the `WS-Discovery` protocol:

Abstract Implementation 2 (Matching with `WS-Discovery`)

$$searchFlightService : \bot \rightarrow (< searchType >, < bookType >) \Rightarrow$$

$$EPRs\_flightReservation \leftarrow probe(QName_{FlightService});$$
$$EPR\_flightReservation \leftarrow selectElement(EPRs\_flightReservation);$$
$$(searchTravel, bookTravel) \leftarrow infer(EPR\_flightReservation);$$
$$return \quad (searchTravel, bookTravel)$$

- **`RESTful`.** We consider `Linked Data` as an example of Web services discovery for `RESTful`. This principle requires knowing a root resource in order to get, by one or multiple links, the desired resources. For our example, this discovery mechanism consists of getting sub-resources of the flight reservation service: getting `URLs` of the `GET` method on the `travel` resource and of the `POST` method on the `booking` resource. In order to avoid the multiple exchanges between a client and a server to get the wanted sub-resources, we consider a registry. The client sends the root-resource `URL` to the registry which will look after the flight reservation sub-resources. In order to abstract this mechanism, we consider a $getSubResources$ operation which requires a $RootResource$ argument, and returns a list of sub-resources description $SubResourcesList$. The $RootResource$ argument here must contain two information: (i) the root `URL` from which the flight reservation resources are accessible and (ii) a semantical information from which the registry knows which sub-resources are wanted. The formalization of this scenario is represented in Figure 7.5.

  The $searchFlightService$ channel of the abstract formalization of Figure 7.2 corresponds here to an operation which calls by default the $getSubResources$ operation using the $rootResource$ description of the flight reservation service. This operation should return a selected sub-resources from the list of type $SubResourcesList$ returned by the search call. The type of the reply channel is more generic than the specific one in Figure 7.2. Therefore, we can deduce the following implementation of the $searchFlightService$ channel for the `RESTful` dynamic discovery:

  Abstract Implementation 3 (Matching with `RESTful`)

$$searchFlightService : \bot \rightarrow (< searchType >, < bookType >) \Rightarrow$$

$$SubResources\_serviceResource \leftarrow getSubResources(rootResource_{FlightService});$$

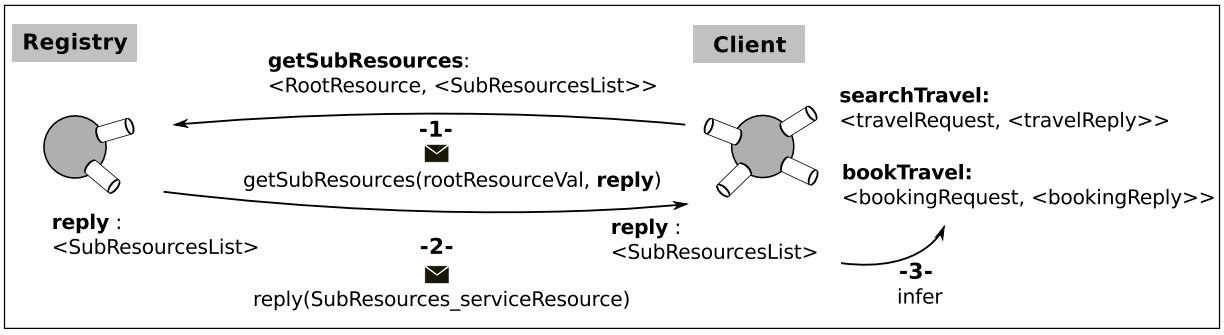


Figure 7.5: Discovery scenario using `Linked Data` for `RESTful` services

$$SubResources\_flightRes \leftarrow selectElement(SubResourcesList\_flightRes);$$

$$(searchTravel, bookTravel) \leftarrow infer(SubResources\_flightRes);$$

$$return(searchTravel, boolTravel)$$

7.2.3 A unified abstraction of the existing discovery protocols

The previous presented formalizations of the different dynamic discovery cases in `SOAP` and `RESTful` show the following common points in the client/registry exchanged data:

1. **A general representation of the required interface**: an interface in the Web services sense is composed of two parts:

   - the typing part: this part represents the input and output types of the different operations in the interface,
   - the access information part: this part represents location access information to an implementation of the typing part.

   A service interface could be defined using one or/and the other part (a UML diagram is represented in Figure 7.6). For instance, a `WSDL` is composed of two parts: a service document which corresponds to the access information part and a binding document which corresponds to the typing part (see the details in Section 2.2.1 of Chapter 2). In `RESTful`, it is possible to query a resource by knowing just its URI and by getting the reply as an `http` content: this is an `Access-Interface` (composed only from the access information part).

   Coming back to the previous presented service discovery examples, we have:

   - In the `UDDI` standard, the $uddiModel$ is globally, a representation of the typing part,
   - In the `WS-Discovery` standard, the $QName$ is a part of the access information part,

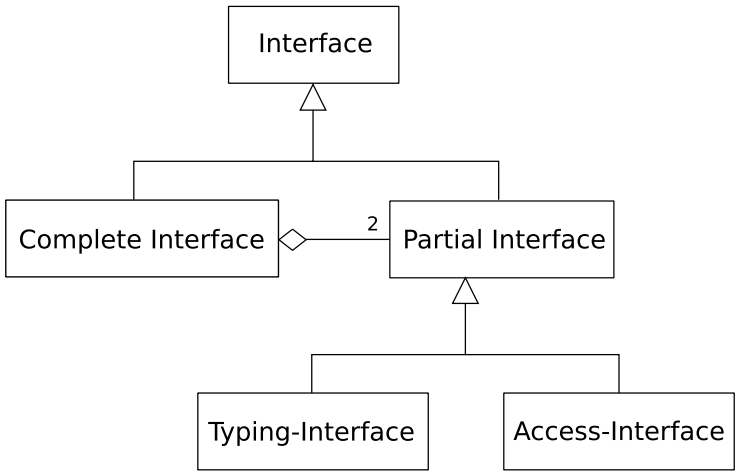


Figure 7.6: Web services interface abstraction

- In the `Linked Data` for `RESTful` discovery, the $RootResource$ is also a part of the access information part.

Thus, conformally to the UML diagram of Figure 7.6, we deduce that all the discovery cases could unify the type of the exchanged interface information using the supertype $Interface$.

2. **A general type for the return values**: As the registry replies its clients with different typed services, a general type is used:

   - $serviceList$ for `UDDI`,
   - $EPRs$ for `WS-Discovery`,
   - $SubResourcesList$ for `Linked Data`.

   This type could be represented as a list of a generic type Θ, to refer to an unspecified service type:

   $$\Theta = \mu X.(\theta, X + End).$$

   We note here that for simplification raisons, our following formalization considers that the return type is simply θ, as we are not interested in how the client or the server makes the choice which service to select from the list.

Based on these two common points, we can deduce a general abstraction of the dynamic service discovery which unifies the existing technologies. This unification is represented in Figure 7.7. The discovery channel, $search$, provided by the registry has the type:

$$search^i : \forall \theta < Interface, < \theta >> .$$

$Interface$ is the general type representing an interface information and θ is a generic type referring to a service type. θ is initialized at runtime when the service required interface is

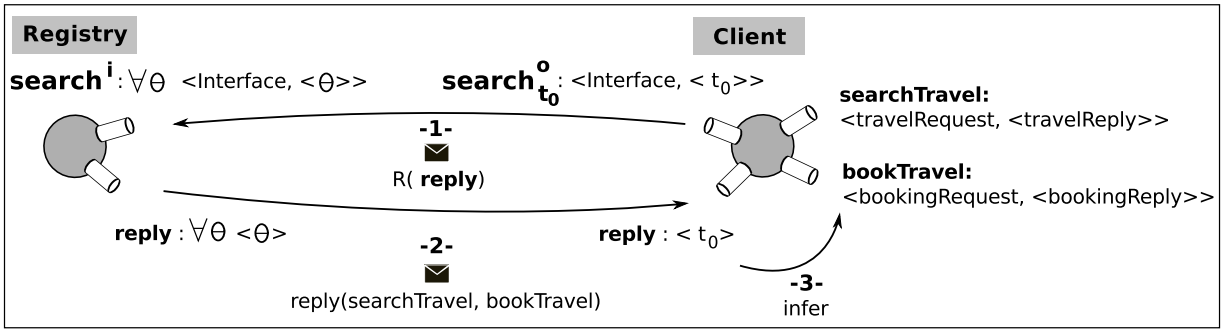


Figure 7.7: An abstract unification of the dynamic discovery mechanisms using generic type

known. At the client side, the $search$ channel will be used to discover the flight reservation service of type t_0, such that t_0 is the type of a couple of channels represented as in the following using the previous defined syntactical sugar:

$$< travelRequest, < travelReply > >, < bookingRequest, < bookingReply > >.$$

The required discovery channel R is thus initialized with the type t_0:

$$search^{o}_{t_0} :< Interface, < t_0 >> .$$

The use of generic typed channels is not a part of our type syntax. Thus, the formalization of the discovery mechanism, as presented in Figure 7.7, needs some reformulations in order to be typed in our formal model.

First, based on the abstract implementation given in Implementation 1, Implementation 2 and Implementation 3 for `UDDI`, `WS-Discovery` and `Atom links` respectively, we can abstract a channel $search_{t_i} :< Interface, < t_i >>$ into a channel $search_i :<< t_i >>$. Thus the abstraction of Figure 7.7 could be simplified as it is represented in Figure 7.8. The $search^{i}_{0}$ channel is typed as in the following:

$$search^{i}_{0} :<< t_0 >>$$

$$\Leftrightarrow$$

$$search^{i}_{0} : < < < travelRequest, < travelReply > >, < bookingRequest, < bookingReply > > > >.$$

Second, our case study for dynamic discovery of services requires that the interfaces required by a client and provided by a service are known and published by a specific registry service (see Section 2.3 of Chapter 2). Based on this hypothesis, we have a fixed set of service types, let us say n number of types. Thus the $search$ channel could be represented as a conjunction of n channels:

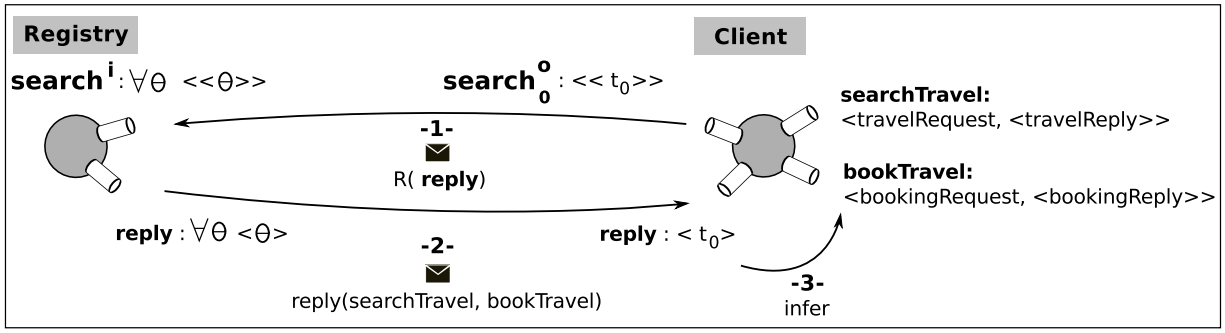


Figure 7.8: A simplified abstract unification of the dynamic discovery mechanisms using generic type

$$search : \forall\theta << \theta >> =$$
$$search_0 :<< t_0 >>$$
$$\wedge$$
$$search_1 :<< t_1 >>$$
$$\wedge$$
$$\ldots$$
$$\wedge$$
$$search_{n-1} :<< t_{n-1} >>$$

The above discussion guides us to a new abstraction well typed in our formal model (see Figure 7.9).

In conclusion, Figure 7.7 and Figure 7.9 are two valid abstractions for the dynamic service discovery. The difference between the two figures is that: the abstraction of Figure 7.7 is faithful to the existing discovery protocols while the abstraction of Figure 7.9 is faithful to our type system.

7.3 A new object-oriented dynamic discovery API

In this section, we show how the previous unified abstraction could be useful to define an OO API for dynamic discovery independently from technical details at the service level. We note here that in the following, we present an abstract implementation of this API adapted to `SOAP`

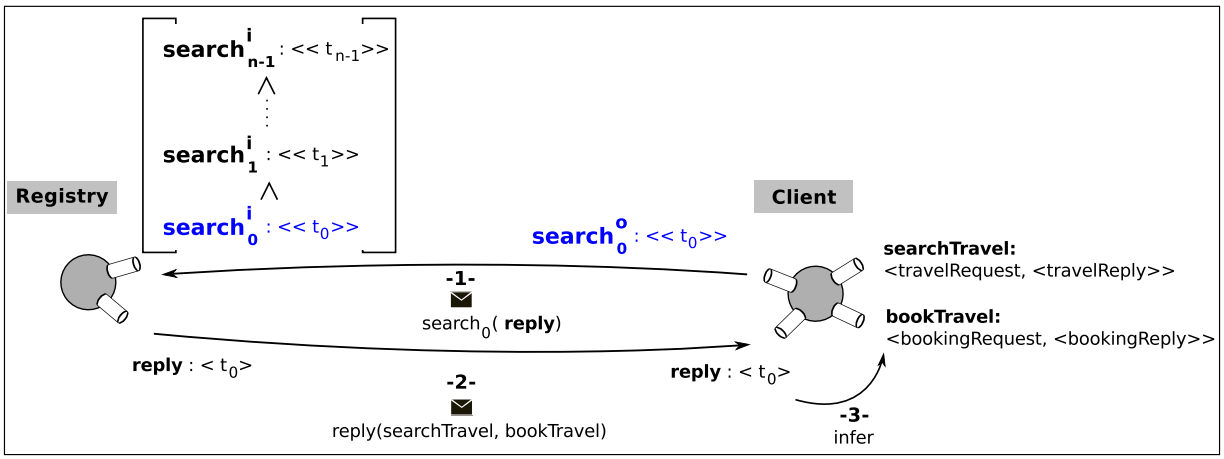


Figure 7.9: A unification of the dynamic discovery mechanism well typed in our formal model

and RESTful. We show in 7.3.1 how this API could be used at the client side. In 7.3.2, we show an abstract implementation of this API by reusing some existing APIs for SOAP and RESTful. All Java codes are not implemented ideas. We use Java here to illustrate these implementations and make the necessary link with the existing methods of the used APIs. Although in the comments of these codes we often refer to the example of "flight reservation" to clarify things, but the code can serve as a general algorithm. Finally, in section 7.3.3, the Java code presented for a new API is independent from the existing APIs and considers interfaces structure to apply subtyping rules. An implementation of these APIs may be subject to future work.

7.3.1 A unified object-oriented interface for dynamic discovery

In order to project the previous unification of dynamic service discovery into the OO environment, we have the choice to consider the abstraction of Figure 7.7 or of Figure 7.9. As generic types could be represented in an OO language like Java, we choose the abstraction of Figure 7.7. The OO API for dynamic discovery must define a search operation which sends an $Interface$ and receives a $Service$. In the OO terms, that corresponds to send an OO interface and to receive an instance of an implementation of this interface which will represent a local proxy for the distant service. The registry operation is defined as in the following, (we use Java language for demonstration):

```
<T> T search(Class<T> interface);
```

The UML diagram of the Registry class is represented in Figure 7.10.

Using this API, a developer who would like to discover a flight reservation service (having MyServiceInt Java interface) and to call its searchTravel operation, has simply to execute the following code:

```
public static void main(String args[]){
 // Instantiating a registry proxy
 Registry registryProxy = new Registry("http://registyURL");
```

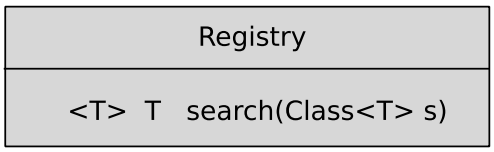


Figure 7.10: UML diagram of the unified Discovery API

```
4
   // Getting the service proxy
6  MyServiceInt proxy =(MyServiceInt) registry.search(MyServiceInt.class);

8  // Calling the searchTravel method
   TravelRequest req = new TravelRequest("Paris", "Berlin", "10/04/2014");
10 List<TravelReply> reply = proxy.searchTravel(req);
}
```

The developer starts by getting an instance of a registry proxy by specifying its `URL`. Then, by calling the search operation on the registry instance, he will get a proxy of the flight reservation service, an instance of an implementation of `MyServiceInt` interface. On the gotten service proxy, the developer could call directly all the required operations of `MyServiceInt`. We consider the following definition of `MyServiceInt` interface:

```
1  public interface MyServiceInt{
   public List<TravelReply> searchTravel(TravelRequest treq);
3  public BookingReply bookTravel(BookingRequest breq)
   }
5
   public class TravelRequest{
7    private String source;
     private String destination;
9    private String date;
     // Getters and Setters
11 }

13 public class travelReply{
     private String id;
15   // Getters and Setters
   }
17
   public class BookingRequest{
19   private String travelId;
     // Getters and Setters
21 }

23 public class BookingReply{
     Boolean confirmation;
25   // Getters and Setters
   }
```

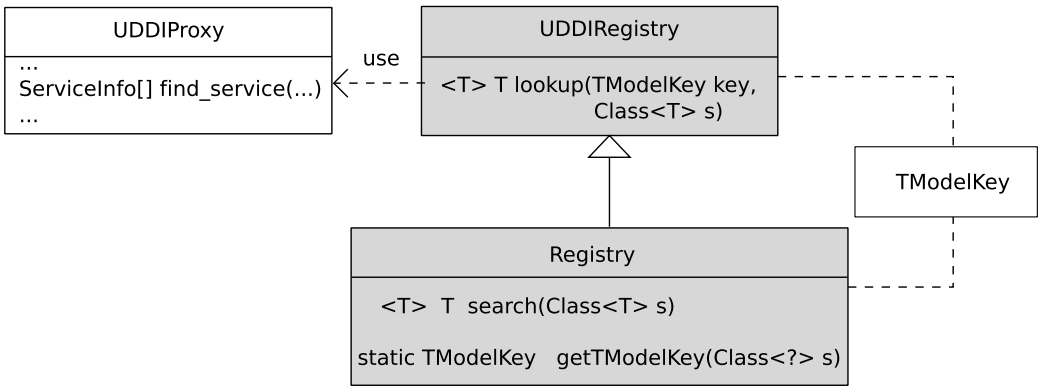


Figure 7.11: UML diagram for improving the UDDI discovery API in Systinet framework

7.3.2 An adapted implementation to existing APIs

In 5.2.1, we have presented some examples about the use of the existing APIs for dynamic discovery in SOAP and RESTful. In the following, we show how these APIs could be reused to build a new API matching with our abstract one. To reach this goal, we come back to the matching done in 7.2.2 between channel searchFlightService of Figure 7.2 and its corresponding concrete once for the existing discovery techniques in Figure 7.3, Figure 7.4 and Figure 7.5. This matching leaded to an abstract implementation to be generalized and concretized in the following. The Java algorithms presented in the following abstract some details: we present here the whole behavior and the main functionalities.

7.3.2.1 An implementation architecture for SOAP

There are two cases:

1. UDDI technology (an application to the Systinet framework):

   Based on Implementation 1, the search operation must call the find_service operation of the UDDI discovery standard. In Chapter 5, we have defined a UDDIRegistry class which has a lookup method. This lookup operation has a complex implementation in order to call the find_service operation of the UDDIProxy class. The UDDIProxy class belongs to the org.idoox.uddi.client.api.v2 package used in Systinet framework. The lookup operation requires a TModelKey instance in arguments. The TModelKey class belongs to org.idoox.uddi.client.structure.v2.tmodel package. Thus, we need to add the UDDIRegistry class to the existing API. Figure 7.11 shows how our Registry class is linked to the exiting classes in Systinet. In this figure, the grey classes are the new classes added to the existing API. Our Registry class inherits from the UDDIRegistry class. Moreover, it defines a static getTModelKey operation which returns a TModelKey instance matching with a service interface.

   Based on this diagram, the matching implementation with UDDI as defined in Implementation 1, is generalized by implementing the search operation of the *Registry* class as in the following:

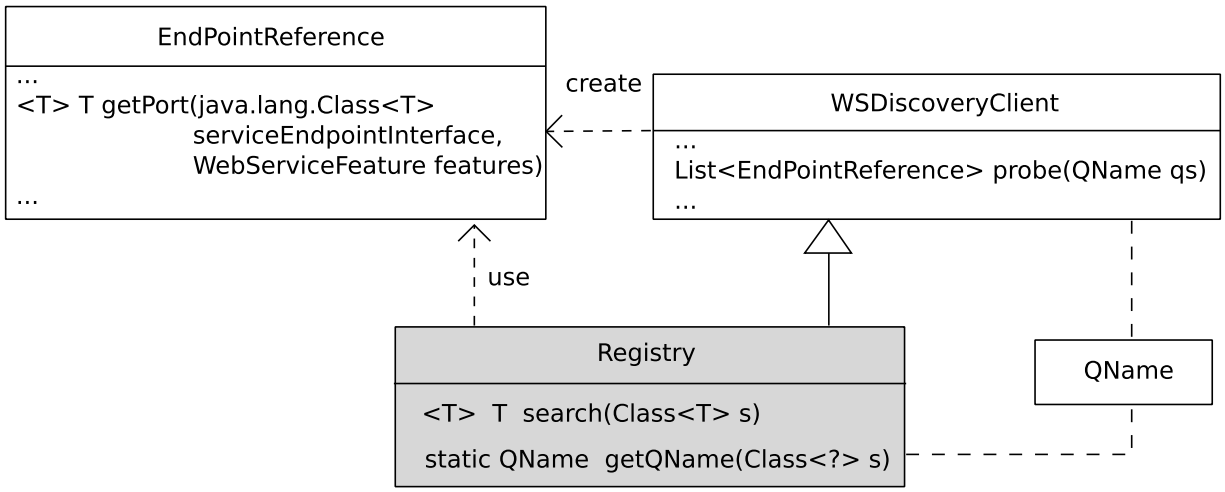


Figure 7.12: UML diagram for improving the `WS-Discovery` API in `cxf` framework

```
public class Registry extends UDDIRegistry{
  public Registry(String URL){
   super(URL);
   }
 <T> T search(Class<T> s){

    // Getting the tModel Key
     TModelKey tModelKey = Registry.getTModelKey(s);

     // Retrieving a service proxy for one of the possibly many
     // services that implement the FlightReservationService
        interface
     T service =  (T) this.lookup(tModelKey, s);
     return service;
  }

  static TModelKey getTModelKey(Class<?> s){
  // this function returns a TModel key corresponding the the
     interface s
  ...
  }
}
```

2. `WS-Discovery technology (an application to the cxf framework):`

Based on Implementation 2, the `search` operation must call the `probe` operation of the `WS-Discovery` standard. The `probe` operation is defined by `WSDiscoveryClient` class which belongs to `org.apache.cxf.ws.discovery` package. The `probe` operation returns a list of `EndPointReference` objects. The `EndPointReference` class be-

longs to `javax.xml.ws` package.

In order to get the service proxy matching with an `EndPointReference`, the `search` operation can call the `getPort` operation of the `EndPointReference` class.

Figure 7.12 shows how our `Registry` class is linked to the exiting classes in `cxf` framework. Our `Registry` class inherits from the `WSDiscoveryClient` class. Moreover, it defines a static `getQName` operation which returns a `QName` instance matching with a service interface.

Based on this diagram, the matching implementation with `WS-Discovery` as defined in Implementation 2 is generalized by implementing the search operation of the `Registry` class as in the following:

```
public class Registry extends WSDiscoveryClient{

  public Registry(String URL){
   super(URL);
  }

 <T> T search(Class<T> s){

  //Getting the QName from the interface
  QName qname = Registry.getQName(s);

  //calling the probe operation using the qname
    List<EndpointReference> references = this.probe(qname);

   //choosing a random service from the gotten list
   EndpointReference epr = random(references);

   //Instantiating a proxy of type FlightReservationInterface
  T proxy = (T) epr.getPort(s, null);

   return proxy;
  }
}
```

7.3.2.2 An implementation architecture for `RESTful`

Based on Implementation 3, the `search` operation must call a `getSubResources` operation to get the sub-resources `Atom links`. For the `RESTful` API, we have to:

- define a `ResourceInterface` interface which has the `getSubResources` operation,
- define a `RootResourceDescription` class, which is the argument of the `getSubResources` operation,
- define a `SubResourceDescription` class which defines the return type of the `getSubResources` operation

- add the appropriated `@LinkResource` on the methods of the `Java` interface.

The `ResourceInterface` is defined as follows:

```
public interface ResourceInterface{
  List<SubResourceDescription> getSubResources(
     RootResourceDescription rootResDesc);
}
```

The `RootResourceDescription` class is defined as follows:

```
@XmlRootElement
public class RootResourceDescription{
  URL url;
  SemanticDescription serviceDescription;
  // getter and setters
}
```

The `SubResourceDescription` class is defined as follows:

```
@Mapped(namespaceMap=@XmlNsMap(jsonName =  "Atom", namespace = "
   http://www.w3.org/2005/Atom"))
@XmlRootElement
public abstract class SubResourceDescription{
  private RESTServiceDiscovery Atom;
  // getter and setter
}
```

`RESTServiceDiscovery` attribute specifies where `Atom links` must be injected

The `SubResourceDescription` class must be abstract because it should not be directly instantiated. Indeed, the operations in the service interface may correspond to CRUD methods on different sub-resources, like for the flight reservation service example: the `searchTravel` operation corresponds to a `GET` on a `travel` resource and `bookTravel` operation corresponds to a `PUT` on a `booking` resource. In order to distinguish to which resource the different CRUD URIs refer, the `Atom links` must distinguish between sub-resources by a specific tag. Thus, each sub-resource must be represented as a inheriting class of the `SubResourceDescription` class. These inherited classes should be instantiated and used to build the corresponding `Atom links`.

The sub-resource classes could not be defined statically because they depend on the service interface. Therefore, they should be created dynamically. It is on the developer to specify the appropriated resources on each service method. For that, we provide a specific annotation:

@Resource(name="resourceName"),

that the developer has to add on the operations of the required object interface.

If we take again the flight reservation service, the required interface at the client side must be annotated as follows:

```
  public interface MyServiceInt{
2 @Get
  @Resource(name="travel")
4 public List<TravelReply> searchTravel(TravelRequest treq);
  @Put
6 @Resource(name="booking")
  public BookingReply bookTravel(BookingRequest breq)
8 }
```

Thus, at runtime, two classes are created for the `travel` and `booking` resources:

```
public class TravelResourceDescription extends SubResourceDescription{
2 public RestServiceDescription getAtom() { return this.Atom;}
}
4 public class BookingResourceDescription extends SubResourceDescription{
  public RestServiceDescription getAtom() { return this.Atom;}
6   }
```

With regards to the `@LinkResource` annotation on the `searchTravel` and the `bookTravel` operations, it should not be defined by the developer. This is because this annotation is linked to the `Atom links` discovery protocol. Thus, at runtime, we need to create a new interface, similar to `MyServiceInt`, and which replaces the `@Resource` annotation with `@LinkResource` as follows:

```
  public interface MyServiceIntCopy{
2 @Get
  @LinkResource(value=TravelResourceDescription)
4 public List<TravelReply> searchTravel(TravelRequest treq);
  @Put
6 @LinkResource(value=BookingResourceDescription)
  public BookingReply bookTravel(BookingRequest breq)
8 }
```

The `@LinkResource` annotations refer to the `TravelResourceDescription` and `BookingResourceDescription` classes.

The `search` operation of the `Registry` class must return a proxy of type `T`. In order to use this proxy directly to call the `searchTravel` and `BookTravel` operations of `MyServiceInt`, the proxy must be built using the static `create` operation of the `ProxyFactory` class (which belongs to `org.jboss.resteasy.client` package) as follows:

```
T proxy = (T) ProxyFactory.create(i, baseURL);
```

such that `i` is of type `Class<T>`, and it refers to the `Java` interface defining `@Path` annotations on its CRUD methods. The `@Path` annotation defines the access `URL` on each method. For instance, `i` corresponds to the following interface in `JAX-RS`:

```
  @Path("/FlightReservationService/")
2 public interface MyServiceIntComplete{
  @Get
4 @Path("travel/")
```

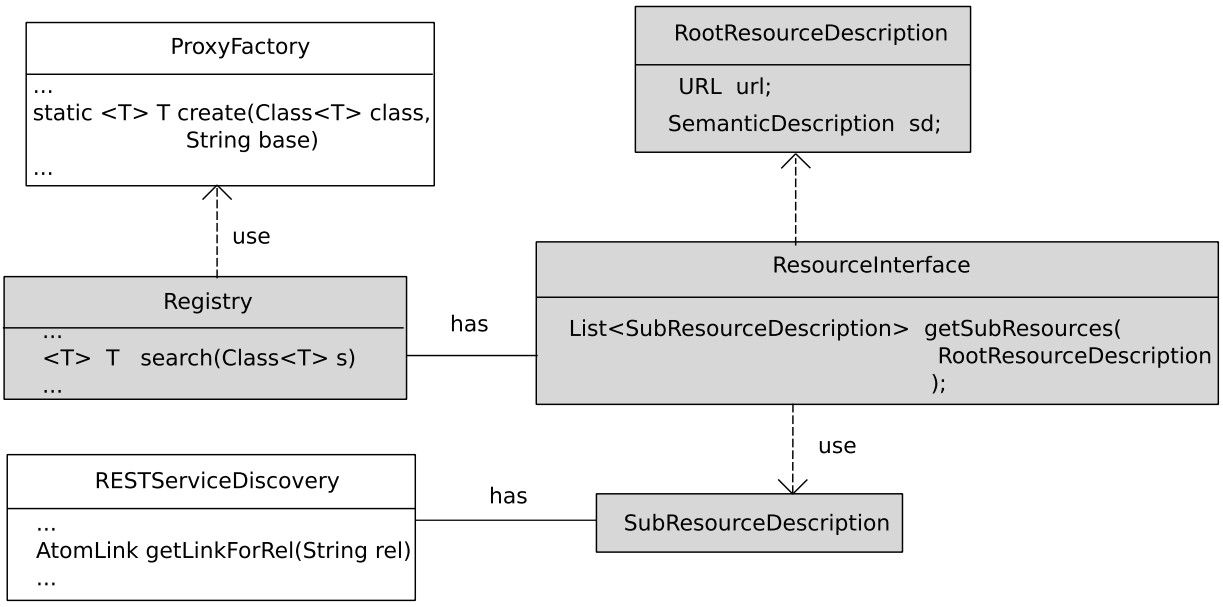


Figure 7.13: UML diagram for improving the `Atom links` discovery for `RESTEasy` framework

```
    public List<TravelReply> searchTravel(TravelRequest treq);
6   @Put
    @Path("booking/")
8   public BookingReply bookTravel(BookingRequest breq)
    }
```

The `baseURL` and the content of the `@Path` annotations is known after calling the `getSubResources` when `Atom links` for the designed service is known. Thus, `MyServiceIntComplete` must be created also at runtime.

According to the previous analysis, we define our `Registry` class with an attribute of type `ResourceInterface`. This attribute is initialized as a proxy to the registry `URL` which provides the `getSubResources` operation. The `search` operation of the `Registry` class starts by creating by reflection dynamically a new interface having the same methods of the required interface. This new interface adds the appropriated `@LinkResource` annotation on all methods as described previously. Then, the search operation calls the `getSubResources` operation of the `proxy` attribute for the `Registry` instance. This call returns a list of `SubResourceDescription` which helps to get the list of `Atom links` for each resource, represented as a `RESTServiceDescription` instance. From the gotten `Atom links`, the search operation get the CRUD paths then it will create dynamically by reflection a new interface extending the required interface and which adds the annotation `@Path` with the appropriated CRUD path value for each method. This new interface, will be used to create the wanted proxy.

Figure 7.13 shows how the architecture of the new discovery API for `RESTful` is in `RESTEasy`. Based on this diagram, the matching implementation with `RESTful Atom links` as defined in Implementation 3, is generalized by implementing the `search` opera-

tion of the *Registry* class as in the following:

```
/** Definition of  Registry  **/

public class Registry{

ResourceInterface proxy;
public Registry(String URL){
  this.proxy = (ResourceInterface) ProxyFactory.create(ResourceInterface.
      class(), URL);
}

<T> T search(Class<T> s){

// Dynamic creation of a new interface having the same methods of
   interface "s" by reflection.
// The new interface adds  the annotation "@LinkResource" for each method
//  in the interface s
Registry.dynamicNewDiscriptionInterface(s);

 List<ResourceDescription> resourceDescList = proxy.getSubResources();
 List<ResourceName, CRUDPaths> pathsList = new ArrayList<ResourceName,
    CRUDPaths>();
 for(ResourceDescription r : resourceDescList){
     RESTServiceDiscovery Atom =  resourceDesc.getAtom();
     // Getting the resource name
     ResourceName resourceName = Atom.getResouceName();
     //Getting the paths for the CRUD methods from the Atom links
     String getPath = Atom.getLinkForRel("self").getHref();
     String putPath = Atom.getLinkForRel("add").getHref();
     String postPath = Atom.getLinkForRel("update").getHref();
     String deletePath = Atom.getLinkForRel("remove").getHref();
     CRUDPaths paths = new CRUDPaths(getPath, putPath, postPath,
        deletePath);
     pathsList.add(resourceName, paths);
 }
 // Dynamic creation of a new interface "i" extending the interface "s" by
     reflection.
// The new interface adds  the annotation "@Path" with the appropriated
   CRUD path value for each method
//  in the interface s
 Class i = Registry.dynamicNewServiceInterface(s, pathsList);

 // Computing the baseURL from the CRUD paths
 String baseURL = Registry.computeBaseURL(getPath, putPath, postPath,
    deletePath);

// creating a proxy from the new complete interface
 T proxy = (T) ProxyFactory.create(i, baseURL);
 return proxy;
  }
```

```
44
    public static Class<?> dynamicNewServiceInterface(Class<?> s, String
        getPath, String putPath, String postPath, String deletePath ){
46  ...
    }
48
    public static Class<?> dynamicNewDescriptionInterface(Class<?> s){
50  ...
    }
52
    public static String computeBaseURL(String getPath, String putPath, String
         postPath, String deletePath ){
54  // Here is a function to compute the base url of the CRUD methods
    }
56  }
```

7.3.3 An object-oriented API for dynamic discovery with subtyping

We present here how the unified OO API for dynamic Web services discovery could be implemented, not only to fit with SOAP and RESTful discovery standards, but also to define a new discovery protocol independent from service model.

Our new discovery protocol has three benefits comparing to the existing ones:

1. Discovery based on service types: Contrary to WS-Discovery standard based on QNames (a qualitative representation of a service type) and to RESTful Atom links based on URLs of root resources, our approach is to discover services based on their explicit types. Indeed, we have proved previously that the structural types for SOAP and RESTful (WSDL and WADL interfaces) could be unified in our formal type system. We benefit from this fact to unify a representation of a service interface based on its type.

2. Discovery based on subtyping: the existing discovery standards lack subtyping in their discovery process. Using the service type to discover a Web service allows to consider subtype services in the search process.

3. Safe discovery based on a weak authentication: Conformally to our formal model, in order to ensure type soundness in presence of malicious agents there is a need to a weak authentication. Such property is not standardized in the existing protocol.

In order to achieve this goal, we need to represent types as values in exchanged messages.

For that, we represent each type as a particular labeled type, as in the following:

Labeled types for type representation	$Type ::= BaseType \mid LabeledType \mid ChannelType \mid IntersectionType \mid UnionType \mid NegationType \mid RecursiveType \mid Variable$
Type for Base type	$BaseType ::= baseType[string], End$
Type for Labeled type	$LabeledType ::= labeledType[label[string], firstType[Type], restType[Type], End], End$
Type for Channel type	$ChannelType ::= channelType[Type], End$
Type for intersection type	$IntersectionType ::= intersectionType[left[Type], right[Type], End], End$
Type for union type	$UnionType ::= unionType[left[Type], right[Type], End], End$
Type for Neg type	$NegationType ::= negationType[Type], End$
Type for Recursive type	$RecursiveType ::= recursiveType[variable[LabeledVariableType], body[Type], End], End$
Type for Variable	$Variable ::= variable[string], End$

For instance, let us consider a case of discovering a service providing two operations: op_1 and op_2, such that: $op_1 : int- > void$ and $op_2 : void- > string$. This service has the following t type:

$$\begin{aligned} t &= op_1 \wedge op_2 \\ &=< in[int], End > \wedge << out[string], End >> \end{aligned}$$

Let us call `I` the required object interface of this service, calling the `search` operation using `I`, generates automatically the following message which should be transmitted to the registry:

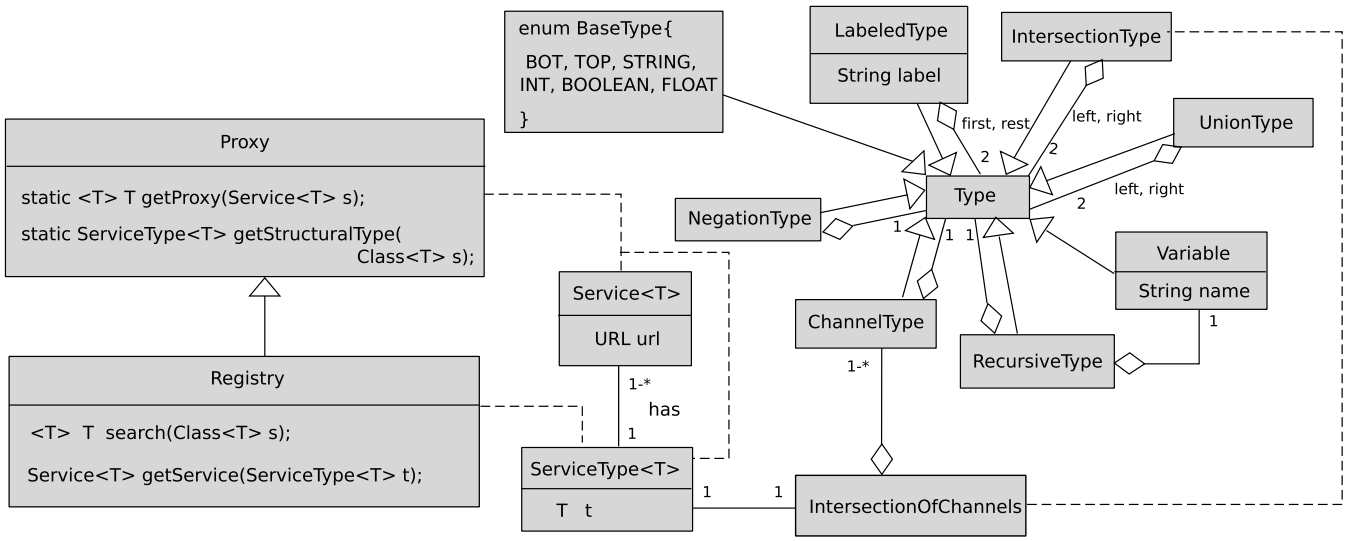


Figure 7.14: UML diagram for the model independent OO API for dynamic discovery

$$
\begin{aligned}
&intersectionType[\\
&\quad left[channelType[labeledType[\\
&\qquad label[\text{"}in\text{"}], firstType[baseType[\text{"}int\text{"}]],\\
&\qquad restType[]\\
&\qquad]\\
&\quad]\\
&]\\
&\quad right[channelType[channelType[labeledType[\\
&\qquad label[\text{"}out\text{"}], firstType[\\
&\qquad baseType[\text{"}string\text{"}]],\\
&\qquad restType[]\\
&\qquad]\\
&\quad]\\
&]\\
&]
\end{aligned}
$$

In order to implement the `Registry` class of the OO API for dynamic discovery, presented in 7.3.1, we have to define from scratch new classes. The UML diagram of Figure 7.14 presents a modeling of our new classes.

In this diagram, we distinguish between two classes: `Service` and `ServiceType`. These classes are generic, parametrized on an interface type `T`:

- `ServiceType<T>` represents a structural interface of the object interface `T`. Conformally to our formal model, a structural interface is an intersection of multiple channels. To represent this link, we associate `ServiceType` class to `IntersectionOfChannels` class. An `IntersectionOfChannels` object refers to an instance of `IntersectionType` class for a set of `channelType` objects. The previous presented labeled types to represent types as values are represented in this diagram by the following classes: `Type`, `BaseType` , `LabeledType`, `IntersectionType`, `UnionType`, `RecursiveType`, `ChannelType` and `Variable`.

- `Service<T>` represents a service reference with a specific `URL` and providing a structural interface associated to an instance of `ServiceType`. A `Proxy` of type `T` could be associated to a `Service<T>` instance using the static method of the `Proxy` class.

The `search` operation of the `Registry` class is defined as follows:

```
public class Registry extends Proxy{
 public Registry(URL url){
   super(url);
 }
 public <T> T search(Class<T> s){
 // getting the structural interface of s
 ServiceType<T> serviceType = Proxy.getStructuralType(s);
 //calling the distant registry server
 Service<T> service = this.getService(serviceType);
 // getting the wanted proxy from the received service instance
 T serviceProxy = (T) Proxy.getProxy(service);
 return serviceProxy;
 }

 public Service<T> getService(ServiceType<T> t){
 // the code here must marshal the ServiceType value to send it to
     the registry
 // and to wait the response to unmarshal it as a Service instance
 ...
 }
}
```

First, we get a `StructuralType` instance corresponding to the interface in parameter of the `search` operation. Then, we ask the registry to get an adequate access `URL` to the service having type `serviceType`. The result is an instance of class `Service<T>`. Finally, the gotten `service` instance could be converted into a proxy of type `T`.

We note here that there is a need to implement the server-side API that support structural subtyping in the `search` operation conformally to the subtyping theoretical rules. Moreover, the `search` operation must take into account the equality between services behavior (or semantic). This is a separate issue that we have not detailed in this thesis.

Finally, in order to apply the weak-authentication algorithm, we choose to make it completely invisible for the developer. At static time, each agent must be a part of a safe zone having a shared secret. This secret is used by a local interceptor which intercept exchanged messages in order to: (i) add the authentication field on the outgoing messages, (ii) checking the authentication field on incoming messages when new channels from the safe zone are discovered.

7.4 Conclusion

In this chapter, we presented a new OO API for Web services discovery that hides from technical details at the service level. To define such an API, we have first showed how the details of the standard interfaces (`WSDL` and `WADL`) could be simplified and abstracted. Then, we have deduced a general abstraction of the existing dynamic discovery protocols based on our formal model. We have showed how this unified API could be implemented by reusing the existing discovery APIs for `RESTful`/`SOAP` services. Another neutral abstract implementation was presented and which works for whatever Web services model. This abstract implementation is based on the principle of using type structures to discover new services. In the client/registry interaction, the message contains an interface representation based on its type. For that, we have used the unified formalization of interfaces in our model. This new implementation idea brings a big advantage compared to the exiting ones: using subtyping in the discovery mechanism.

8

A New Specification for Data Binding

Contents

As their name suggests, the OO frameworks for Web services should offer to their developers an OO facet to develop Web services. This facet should be compliant with the OO development practices and keeps the developers far from low level Web services details. We refer here more particularly to allow subtyping using the substitution principle and to abstract from marshalling/unmarshalling details in what concerns the schema binding.

In Chapter 5, we showed that the existing `cxf` framework does not respect this principle:

1. interoperability fails when the substitution principle is applied in its two forms: value substitution and interface substitution,
2. JAXB schema binding annotations are mandatory at the object level and weaken the loose coupling between the object level and the the service level.

This chapter 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 presenting a concrete implementation in the cxf framework.

Precisely, we provide the following contributions:

- In Section 8.1, based on the architecture of an OO framework for Web services as presented in 3.1, we propose a new specification founded on commutative diagrams of the data binding involved in the framework. Then, we show that the new specification solves the issues and can be concretely implemented in cxf using JAXB data binding.
- In Section 8.2, we show how the schema bindings could be completely abstract for the developer using the JAXB data binding. We present then how cxf can be configured to use JAXB in order to ensure a loose coupled schema binding at the object level.
- We finish by combining the two proposed solutions (for the substitution problem and loose coupled schema binding) to propose a standard configuration of cxf for RESTful and SOAP cases, in Section 8.3. We discuss the advantages brought by this combination and we discuss the performance of our proposed solution.

8.1 An adaptation for an interoperability by subtyping

In this section, we first discuss a problem analysis based on a formal abstraction of data binding functions. Then, we propose an abstraction in commutative diagrams in order to deduce a specification resolving the existing issues. Based on this specification, we propose then a solution adapted to cxf framework.

8.1.1 Problem analysis

In the following, we first present an abstraction representing how the OO framework drives the data binding in order to call its different functions. Second, using this abstraction, we show how we are able to understand the cause of the interoperability problems in the cxf framework due to the substitution principle.

8.1.1.1 Driving the data binding

In 3.1, we have presented four essential functions of data binding: marshalling/unmarshalling and schema generation/schema compilation. Here, we show a formalization of these functions. This formalization abstracts the way an OO framework for Web services calls these functions, based on the control flow diagram presented in 3.1.5.

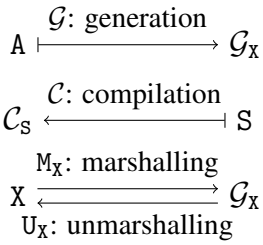


Figure 8.1: Driving the Data Binder

Schema generation. The framework provides an object type, and consequently its associated tree of used types. The data binder 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 8.1 as a function over types: function $\mathcal{G}$ maps object type X to schema $\mathcal{G}_X$.

Schema compilation. The framework provides a schema. The data binder 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 8.1 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 data binder returns a document representing an observation of the object. This observation is based on calls to getters (depending from the specified schema mapping), 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 could be the dynamic type of the object, determined in `Java` by a call to method `getClass`, or 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. At marshalling, the framework has the choice to activate or not the validation of the output document. This validation depends on the associated structural type to the object type. The marshalling function is denoted as a family of functions M_X indexed over object type X and defined from object type X to schema $\mathcal{G}_X$, as represented in Figure 8.1.

Unmarshalling. The framework provides a document. The data binder returns an object such that its marshalling produces the document given as input. In general, 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, one extra parameter is required: an object type used to construct the returned object. The object type is determined statically, from

the declaration of the service. At unmarshalling, the framework has the choice to activate or not the validation of the input document. This validation depends on the associated structural type to the object type. The unmarshalling function is denoted as a family of functions $\mathtt{U}_{\mathtt{X}}$ indexed over object type X and defined from schema $\mathcal{G}_{\mathtt{X}}$ to object type X, as represented in Figure 8.1.

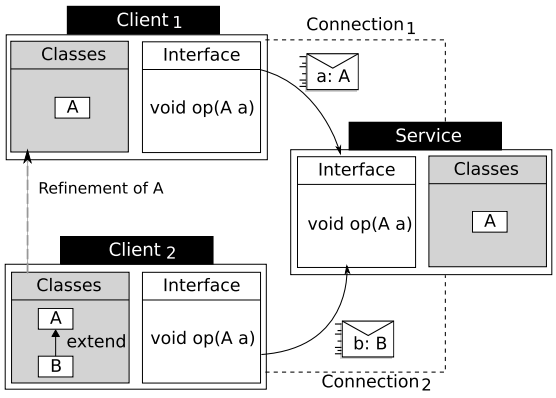
To conclude, we can now formalize the definition of equivalence between objects given in an informal way in 3.1.2 (see Definition 1).

Definition 6 (Equivalence for Marshallable Types) *Given two marshallable types* A *and* B*, we say that* A *is equivalent to* B*, and write* $\mathtt{A} \equiv \mathtt{B}$*, when* $\mathcal{G}_{\mathtt{A}} = \mathcal{G}_{\mathtt{B}}$*. Assume* $\mathtt{A} \equiv \mathtt{B}$*; if object* a *has type* A *and object* b *has type* B*, we say that* a *is equivalent to* b *when* $\mathtt{M}_{\mathtt{A}}(\mathtt{a}) = \mathtt{M}_{\mathtt{B}}(\mathtt{b})$.

8.1.1.2 Revisited scenarios

Based on the previous control flow abstraction between the framework and the data binder, we revisit in the following the scenarios presented in 5.2.2 in order to localize the cause of the detected errors.

Value substitution: We bring again here the scenario example and test results in Figure 8.2. 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`). As it is shown in Figure 8.3(a), the received document root element, `b`, differs from the expected one, `a`. Thus it is not possible to proceed the unmarshalling using `A` object type. The error is due to the non-equivalence between both types (see Definition 6). In the `RESTful` case for the `cxf` version `2.7.5` (see Figure 8.3(b)) and in the `SOAP` case (see Figure 8.3(c)), the static type, `A` 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.

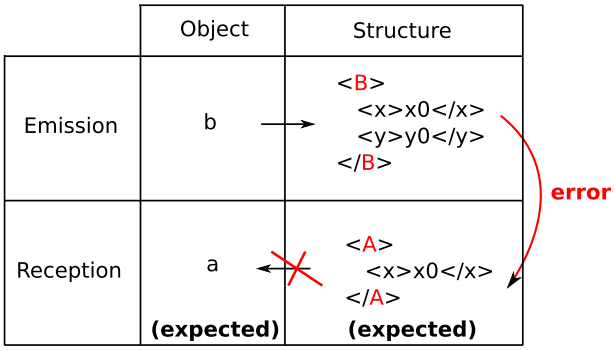


(a) Example

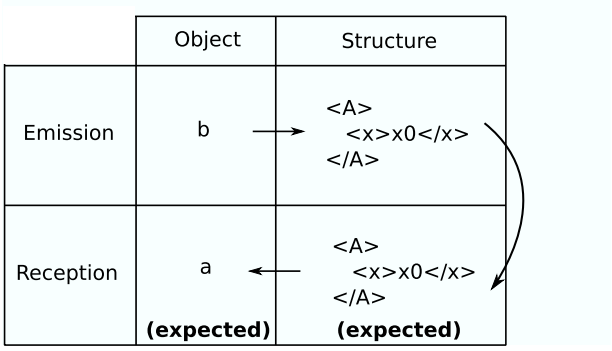
	SOAP	RESTful
Interoperability is preserved	**Yes**	No

(b) Test results with `cxf` 2.5.10

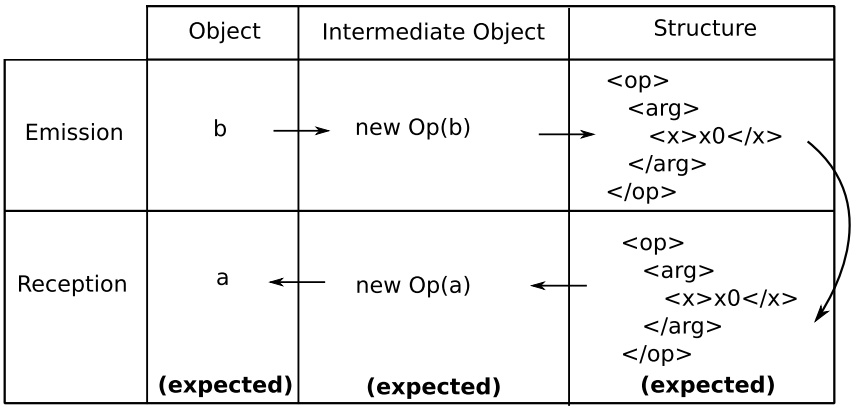
Figure 8.2: Value substitution



(a) `RESTful`: marshalling with dynamic type B at emission in `cxf` 2.5.x



(b) `RESTful`: marshalling with static type A at emission in `cxf` 2.7.x



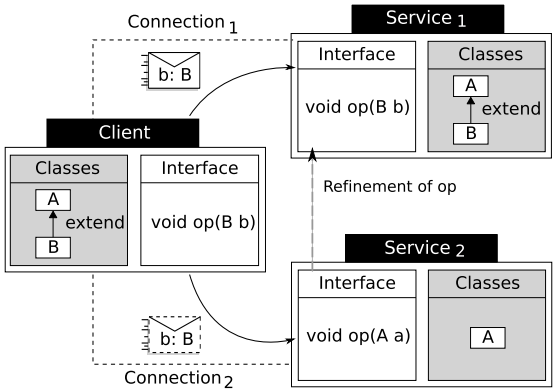
(c) `SOAP`: marshalling/unmarshalling with static type A

Figure 8.3: Exchanges between emission and reception for the value substitution scenario

Interface substitution: We bring again here the scenario example and test results in Figure 8.4. For the `RESTful` case, as the service operation `op` has `B` as an input argument at emission, the marshalling type of the used `B` instance to call `op` is `B`. At reception, `A` (the supertype of `B`) is only known and it is the unmarshalling object type. Thus, the scenario reproduces a similar exchange to the value substitution scenario for `cxf` 2.5.x as represented in Figure 8.3(a). The thrown exception is thus due to the same reason explained before. For the `SOAP` case, the error is due to a different reason. Figure 8.5 shows the exchanges between emission and reception when `A` and `B` are not equivalent. The received structure has the correct root element, `op`, which allows to proceed in the unmarshalling using the `Op` object type. However, the fault comes from the validation of the input document, conforming to the structural type associated to the subtype, against the structural type associated to the supertype when `A` and `B` are not equivalent.

Discussion. It was possible to force the subtype, `B` and its supertype `A` 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.



(a) Example

		SOAP	RESTful
Interoperability is preserved	B ≡ A	**Yes**	No
	B ≢ A	No	No

(b) Test results with `cxf`

Figure 8.4: Interface substitution

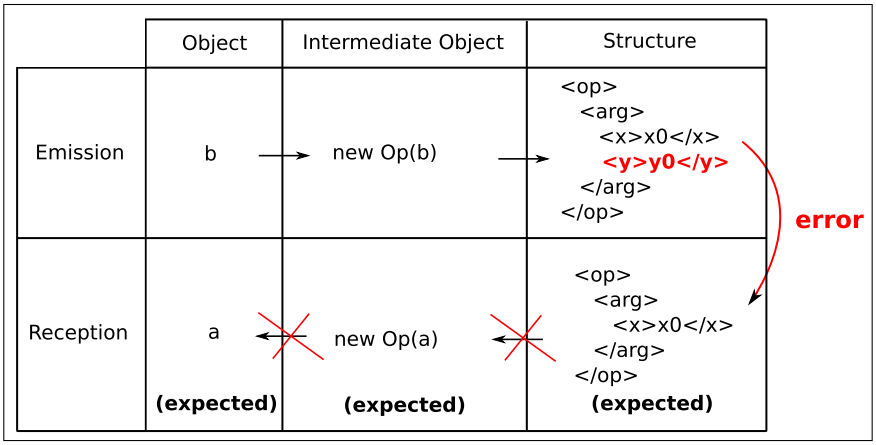


Figure 8.5: `SOAP` case of the interface substitution scenario: marshalling using B type and unmarshalling using A type, (B ≢ A)

8.1.2 An abstraction in commutative diagrams

We now revisit both scenarios, value substitution and interface substitution, while generalizing them, and propose new requirements for 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 3.1.4: 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 diagrams are `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 binders 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:

- avoiding all the faults analyzed in 8.1.1.2,
- 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.

8.1.2.1 From diagrams to requirements: The specification

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 5.2.2, 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.*

$$\forall\, \mathtt{A} \,.\, \mathtt{U}_\mathtt{A} \,;\, \mathtt{M}_\mathtt{A} = \mathrm{id}_{\mathcal{G}_\mathtt{A}}.$$

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.

Revisiting the first scenario involving a value substitution (see Figure 8.2(a)): 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.

$$\begin{array}{ccc} \mathcal{G}_\mathtt{A} & \xrightarrow{\mathtt{U}_\mathtt{A}} & \mathtt{A} \\ \vdots & & \\ \mathtt{B} & \xrightarrow[\mathtt{M}_\mathtt{B}]{} & \mathcal{G}_\mathtt{B} \end{array} \qquad \begin{array}{ccccc} \mathcal{CG}_\mathtt{A} & \xrightarrow{\mathtt{M}_{\mathcal{CG}_\mathtt{A}}} & \mathcal{G}_\mathtt{A} & \xrightarrow{\mathtt{U}_\mathtt{A}} & \mathtt{A} \\ \mathtt{i}\uparrow & & & & \\ \mathtt{B} & & & & \end{array}$$

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*[1] *and the marshalling function* M *is a natural transformation: given any object types* A *and* B*, for any (interesting*[2]*) function f : B $\to$ A, there exists a function $\mathcal{G}_f : \mathcal{G}_\mathtt{B} \to \mathcal{G}_\mathtt{A}$, the lifting of f, such that the following diagram commutes.*

$$\begin{array}{ccc} \mathtt{A} & \xrightarrow{\mathtt{M}_\mathtt{A}} & \mathcal{G}_\mathtt{A} \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle \mathcal{G}_f} \\ \mathtt{B} & \xrightarrow[\mathtt{M}_\mathtt{B}]{} & \mathcal{G}_\mathtt{B} \end{array}$$

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

$$\begin{aligned} \mathtt{M}_\mathtt{B} ; \mathcal{G}_f &= f ; \mathtt{M}_\mathtt{A}, \\ \mathtt{U}_\mathtt{B} ; \mathtt{M}_\mathtt{B} ; \mathcal{G}_f &= \mathtt{U}_\mathtt{B} ; f ; \mathtt{M}_\mathtt{A}, \\ \mathcal{G}_f &= \mathtt{U}_\mathtt{B} ; f ; \mathtt{M}_\mathtt{A}. \end{aligned}$$

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 $\mathcal{G}\mathcal{C}\mathcal{G}_\mathtt{A} = \mathcal{G}_\mathtt{A}$).

$$\begin{array}{ccccc} \mathcal{C}\mathcal{G}_\mathtt{A} & \xrightarrow{\mathtt{M}_{\mathcal{C}\mathcal{G}_\mathtt{A}}} & \mathcal{G}_\mathtt{A} & \xrightarrow{\mathtt{U}_\mathtt{A}} & \mathtt{A} \\ {\scriptstyle \mathtt{i}}\uparrow & & \uparrow{\scriptstyle \mathcal{G}_\mathtt{i}} & & \\ \mathtt{B} & \xrightarrow[\mathtt{M}_\mathtt{B}]{} & \mathcal{G}_\mathtt{B} & & \end{array}$$

Let us revisit the second scenario involving an interface substitution (see Figure 8.4(a)): 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:

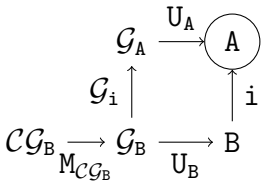
$$\begin{array}{ccc} & \mathcal{G}_\mathtt{A} \xrightarrow{\mathtt{U}_\mathtt{A}} \mathtt{A} & \\ & \vdots\uparrow & \\ \mathcal{C}\mathcal{G}_\mathtt{B} & \xrightarrow[\mathtt{M}_{\mathcal{C}\mathcal{G}_\mathtt{B}}]{} \mathcal{G}_\mathtt{B} & \end{array}$$

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

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

[2] 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.

sensible notion of equivalence between the initial object and the final object. Definition 6 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 $\mathtt{M}_{\mathcal{CG}_{\mathtt{B}}} ; \mathcal{G}_{\mathtt{i}} ; \mathtt{U}_{\mathtt{A}} ; \mathtt{M}_{\mathtt{A}} = \mathtt{M}_{\mathcal{CG}_{\mathtt{B}}} ; \mathcal{G}_{\mathtt{i}} ; \mathtt{U}_{\mathtt{A}} ; \mathtt{M}_{\mathtt{A}}$, which can be directly deduced from Core Requirement 2. But we can also propose another definition, more accurate. 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 both cases, in other words, 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 $f : X \to \mathtt{A}$ and $\mathtt{g} : X \to \mathtt{A}$, their equivalence in the diagram means the equation $f ; \mathtt{M}_{\mathtt{A}} = \mathtt{g} ; \mathtt{M}_{\mathtt{A}}$ instead of $f = \mathtt{g}$. Commutativity therefore means here:

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

which can be deduced from $\mathtt{U}_{\mathtt{A}} ; \mathtt{M}_{\mathtt{A}} = \mathrm{id}_{\mathtt{A}}$ and $\mathcal{G}_{\mathtt{i}} = \mathtt{U}_{\mathtt{B}} ; \mathtt{i} ; \mathtt{M}_{\mathtt{A}}$.

The `SOAP` case. With `SOAP`, the sequence of transformations from the client to the server is complex during a call `op(a)` since it involves a call reification, as recalled in Figure 3.8. 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 8.6 sums up these definitions.

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

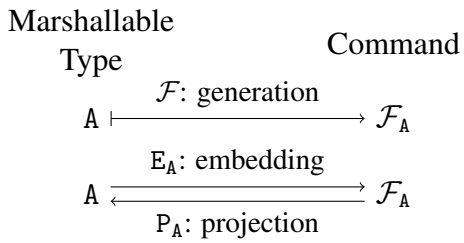


Figure 8.6: Command Generation

For a type $\mathtt{A}$ 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:*

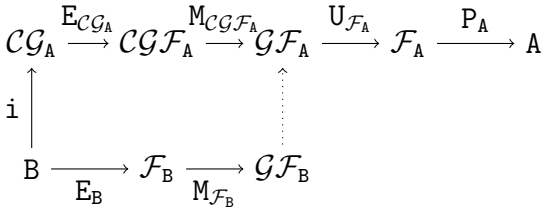
$$\forall\, \mathtt{A}, \mathtt{B} \,.\, \mathcal{F}_{\mathtt{A}} = \mathcal{F}_{\mathtt{B}} \Rightarrow \mathtt{A} = \mathtt{B}.$$

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:*

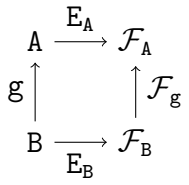
$$\forall\, \mathtt{A} \,.\, (\mathtt{P}_{\mathtt{A}} ; \mathtt{E}_{\mathtt{A}} = \mathrm{id}_{\mathcal{F}_{\mathtt{A}}}) \wedge (\mathtt{E}_{\mathtt{A}} ; \mathtt{P}_{\mathtt{A}} = \mathrm{id}_{\mathtt{A}}).$$

We now come to the new requirements dealing with subtyping. Consider again the first scenario, involving a value substitution (see Figure 8.2(a)). The execution can be pictured as follows.



Besides the execution path starting with a conversion from $\mathtt{B}$ to $\mathcal{CG}_{\mathtt{A}}$ via $\mathtt{i}$, 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.

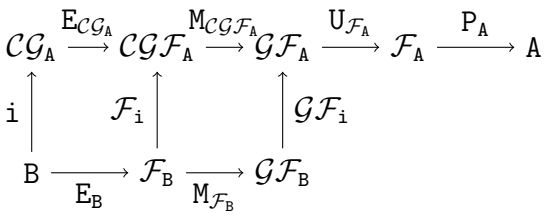
New Requirement 7 (Lifting for Command Generation) *The command generation $\mathcal{F}$ is a functor and the embedding function $\mathtt{E}$ is a natural transformation: given any object types $\mathtt{A}$ and $\mathtt{B}$, for any (interesting) function $\mathtt{g} : \mathtt{B} \to \mathtt{A}$, there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



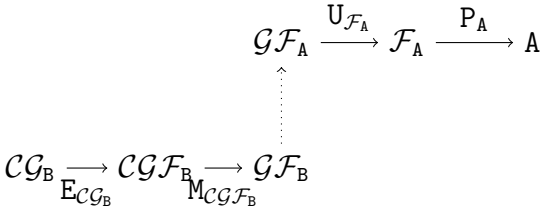
As for the schema generation, the lifting $\mathcal{F}_{\mathtt{g}} : \mathcal{F}_{\mathtt{B}} \to \mathcal{F}_{\mathtt{A}}$ is effectively computable:

$$\mathcal{F}_{\mathtt{g}} \quad = \quad \mathtt{P_B} \,;\, \mathtt{g} \,;\, \mathtt{E_A}.$$

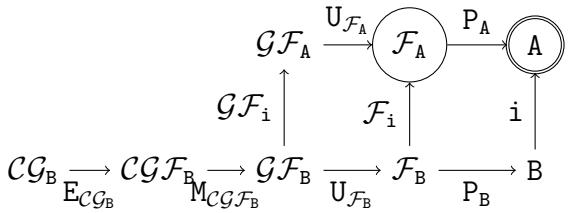
We can now deduce that the following diagram is commutative: the choice between static types and dynamic types no longer matters.



Let us consider the second scenario involving an interface substitution (see Figure 8.4(a)). 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 $f : X \to \mathcal{F}_{\mathtt{A}}$ and $\mathtt{g} : X \to \mathcal{F}_{\mathtt{A}}$, their equivalence in the diagram means the equation $f\,;\mathtt{M}_{\mathcal{F}_{\mathtt{A}}} = \mathtt{g}\,;\mathtt{M}_{\mathcal{F}_{\mathtt{A}}}$, and for two paths $f : X \to \mathtt{A}$ and $\mathtt{g} : X \to \mathtt{A}$, their equivalence in the diagram means the equation $f\,;\mathtt{E}_{\mathtt{A}}\,;\mathtt{M}_{\mathcal{F}_{\mathtt{A}}} = \mathtt{g}\,;\mathtt{E}_{\mathtt{A}}\,;\mathtt{M}_{\mathcal{F}_{\mathtt{A}}}$. Thus, the final objects that are computed are equivalent, as for the `RESTful` case.

Summary. Finally, with all the requirements defined in this section, the substitution principle can be recovered, in a robust way. All the problems presented in our studied scenarios, 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 $\mathtt{i}$ is the canonical conversion from $\mathtt{B}$ to $\mathtt{A}$, the liftings are defined as follows, in the `RESTful` case and in the `SOAP` case respectively:

$$\mathcal{G}_{\mathtt{i}} = \mathtt{U}_{\mathtt{B}}\,;\mathtt{i}\,;\mathtt{M}_{\mathtt{A}}, \tag{8.1}$$

$$\mathcal{GF}_{\mathtt{i}} = \mathtt{U}_{\mathcal{F}_{\mathtt{B}}}\,;\mathtt{P}_{\mathtt{B}}\,;\mathtt{i}\,;\mathtt{E}_{\mathtt{A}}\,;\mathtt{M}_{\mathcal{F}_{\mathtt{A}}}. \tag{8.2}$$

Equations 8.1 and 8.2 could be generalized as a property for every OO framework to ensure the substitution principle, for the following reasons:

- the principle of commutative diagrams based on ensuring equivalence between objects at emission and at reception is a property to be respected everywhere,
- the previous diagrams abstract the data flow and the control flow for whatever OO framework. Indeed, a framework can differ from another one by the marshalling/unmarshalling mechanisms (the form of the document structures and the way they are converted from and to objects), which are abstracted in our commutative diagrams.

Therefore, the core and new requirements specified previously should be respected by all OO frameworks for Web services and from which we deducted the computation of the lifting functions.

8.1.2.2 A concretization of the specification

We now pave the way towards a concretization of the lifting functions used for conversion, following Equations 8.1 and 8.2. Our aim is to conclude a standardization which should be respected in every OO framework in order to allow interoperability by subtyping. This standard solution should be fully integrated into the framework in a transparent way for the user of the framework.

Meaning of the Specification. The specification given by Equations 8.1 and 8.2 can be interpreted as follows. For the `RESTful` case, on input $\texttt{doc}_\texttt{B}$ with type $\mathcal{G}_\texttt{B}$, 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 $\texttt{doc}_\texttt{A}$ with type $\mathcal{G}_\texttt{A}$. For the `SOAP` case, on input $\texttt{doc}_{\mathcal{F}_\texttt{B}}$ with type $\mathcal{GF}_\texttt{B}$, the framework must successively call (i) the unmarshalling function by passing type $\mathcal{F}_\texttt{B}$, produced by the command generation from `B`, (ii) the projection function, corresponding to the call to a getter of command class $\mathcal{F}_\texttt{B}$, (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 $\mathcal{F}_\texttt{A}$ and to a setter, (v) the marshalling function by passing type $\mathcal{F}_\texttt{A}$ and get a document $\texttt{doc}_{\mathcal{F}_\texttt{A}}$ with type $\mathcal{GF}_\texttt{A}$.

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. For a standard data binding, like `JAXB`, a document marshalled through a subtype, $\texttt{doc}_\texttt{B}$, differs from a document marshalled through a supertype, $\texttt{doc}_\texttt{A}$, in two points:

- the name of the root tag, which normally refers to the schema binding of the corresponding object type,
- the presence of additional elements due to the mapping of additional attributes defined in the subtype.

In order to apply this algorithm, the schema binding must define fixed tag names for the subroot elements when marshalling recursively the nested objects, independently from their marshalling object types. We require this condition in order to reduce the complexity of the lifting algorithm by reducing the gap between structures associated to subtypes and those associated to their supertypes. The `JAXB` data binding, for example, satisfies this condition. In order to illustrate this idea, we consider the example of Figure 8.7. In this example we consider a marshalling of `eMsg` instance of `A'` through type `A'` at emission and an unmarshalling thought type `A` supertype of `A'` at reception. We consider a nested object, `c'` in `eMsg`, an instance of `C'`, marshalled through type `C'` at emission and unmarshalled through type `C` supertype of `C'` at reception. The example shows how the tag associated to marshalling the nested object `c'` does not depend from the name of the marshalling type, `C'`, but from the schema binding of the attribute defined in class `A`, here it is `x`. Thus, the sent document differs from the expected one with:

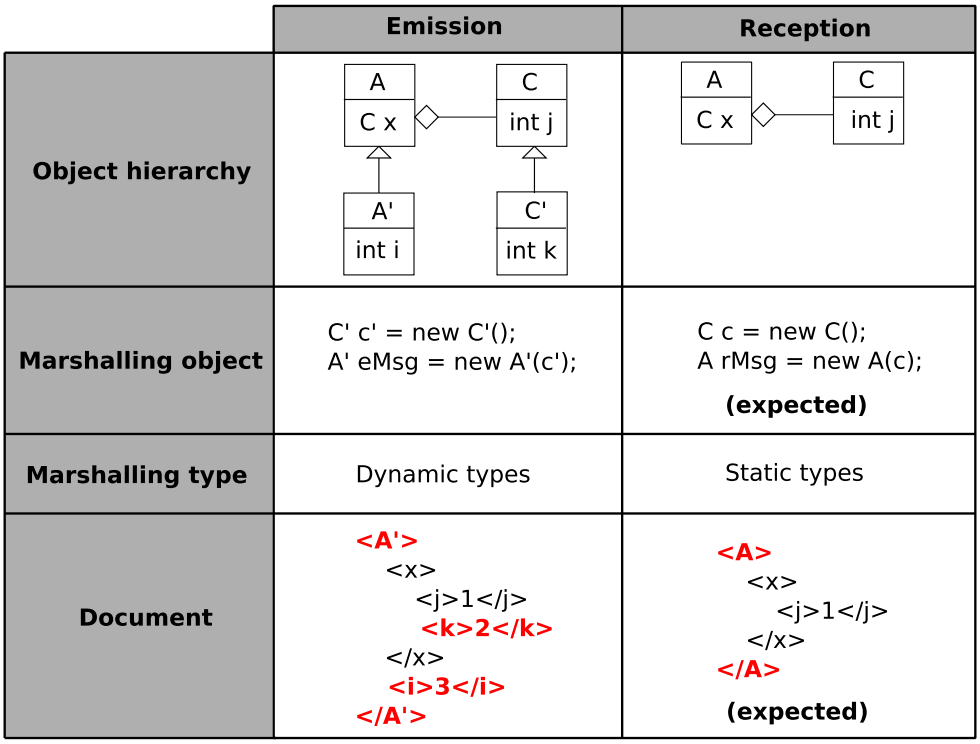


Figure 8.7: An example of interoperability by subtyping with nested objects

- the root tag name (`A'` instead of `A`)
- the presence of two additional elements:
  - `k` as subelement of `x`, derived from marshalling the `k` attribute of `c'`,
  - `i` as subelement of `a'`, derived from marshalling the `i` attribute of `eMsg`.

Hence, we deduce the following lifting algorithms for:

- **`RESTful`:** in order to transform document $\texttt{doc}_B$ into document $\texttt{doc}_A$ with type $\mathcal{G}_A$, it suffices to rename if needed the root tag, and to extract if possible the sub-documents of $\texttt{doc}_B$ to match the definition of type $\mathcal{G}_A$. This extraction corresponds to a well-studied algorithm, the `tree inclusion problem` [13],
- **`SOAP`:** in order to transform document $\texttt{doc}_{\mathcal{F}_B}$ into document $\texttt{doc}_{\mathcal{F}_A}$ with type $\mathcal{GF}_A$, it suffices to extract if possible the sub-documents of $\texttt{doc}_{\mathcal{F}_B}$ to match the definition of type $\mathcal{GF}_A$. Contrary to the `RESTful` case, there is not a need to rename the root tag because in `SOAP` this tag refers to the called service operation. If the root tag does not match with the expected one thus an error must be detected because the service is enable to identify the called operation.

Deployment of the implementation. 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 apply our algorithm: it is at reception at unmarshalling.

Two implementations are possible for the lifting algorithm:

- **Implementation 1:** inside the unmarshalling function of the data binder,
- **Implementation 2:** inside the OO framework by an interception mechanism which applies the necessary modifications on the received document before sending it to the data binder for unmarshalling.

Each implementation has an advantage comparing to the other one:

- The advantage of the first implementation comparing to the second one lies in the performance cost. In the first implementation, renaming the root element could be at the moment this element is treated by the data binder for unmarshalling. The same for the additional elements, they will be ignored by the data binder when they do not match with the expected elements. Therefore, the document tree is navigated one time to apply both lifting and unmarshalling algorithms. While, in the second implementation, as the lifting algorithm is implemented outside the data binder, there is a need to navigate the input document twice: one for lifting and one for unmarshalling.
- The advantage of the second implementation comparing to the first one lies in the implementation cost. The first solution implies modifying the existing data binders, (at least one data binder per document language) or defining from scratch a new data binder in order to apply the lifting algorithm at unmarshalling. Moreover, this data binder should support unmarshalling with projection and renaming the document root element or just a simple projection in order to be compatible with the lifting algorithm applied to `RESTful` and `SOAP` respectively (as we described before). While, by modifying the OO framework, the implementation of the lifting algorithm is done once for all independently from the plugged data binder.

Finally, choosing one or other solution depends on each context and on the developer preferences. In the following, we present which is the less costly solution for the `cxf` framework.

8.1.3 A light solution adapted to `cxf`

In order to make the right implementation choice for `cxf`, we bring again here the overview of the `cxf` architecture based on inbound and outbound chains previously presented in 3.1.5 (see Figure 8.8). Moreover, as the lifting algorithm should be applied at unmarshalling, we remind also of the UML diagram for the `Unmarshal` phase (see Figure 8.9).

Based on Figures 8.8 and 8.9, we discuss the two implementations previously described, `Implementation 1` and `Implementation 2`, and we choose the more convenient one in cost of implementation and performance for `cxf`.

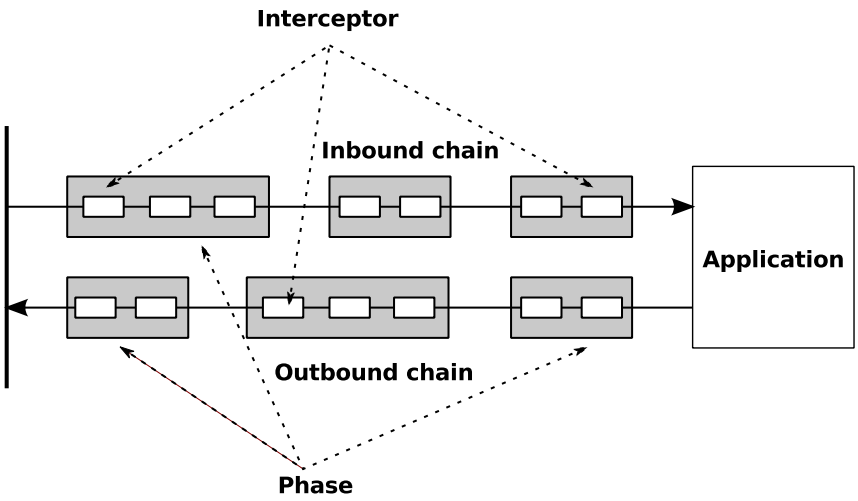


Figure 8.8: Inbound and outbound chains in `cxf`

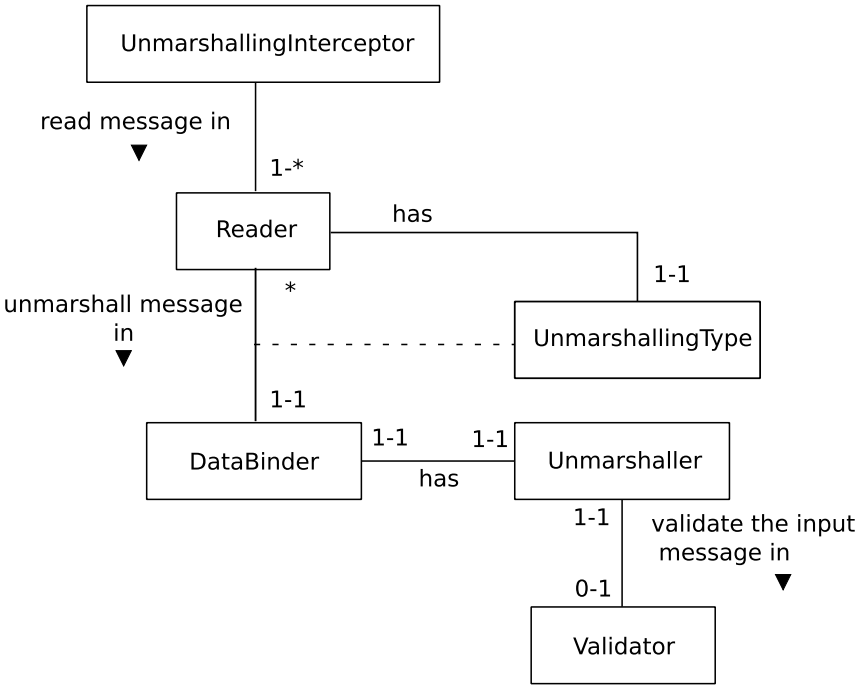


Figure 8.9: Abstract UML of the unmarshalling phase in `cxf`

Implementation 1. Modifying the unmarshalling method of the data binder affects three classes conformally to the UML diagram in Figure 8.9:

- the `DataBinder`: it should be replaced by the new one
- the `Unmarshaller`: it should apply the lifting algorithm
- the `Reader`: it should be associated to the new data binder and it is on it to decide which behavior the data binder should have for unmarshalling (a projection with renaming the root element or without) depending from the used service technology, `RESTful` or `SOAP`.

Thus, there is a need to define two readers, one for each service technology. Moreover, the data binder should at least handle the two most known document formats and which are used by `cxf`, (`XML` and `JSON`).

As `JAXB` is the default data binder used by `cxf` for `RESTful` and `SOAP`, and which is able to handle both document formats (`XML` and `JSON`), we can reduce the implementation cost by considering it as our studied data binder. This way, we can reuse the existing readers with some adaptations.

Implementation 2. Thanks to the `cxf` flexible architecture as defined in Figure 8.8, it is relatively easy to add an interceptor to the interceptor chains. Thus, the lifting algorithm could be implemented as an interceptor which should be added to the `Unmarshal` phase in the inbound chain before the `UnmarshallingInterceptor` actually implemented in the `Unmarshal` phase.

The less costly implementation. The drawback of the implementation inside the data binder (`Implementation 1`), which is due to the implementation cost, is reduced for the `cxf` framework thanks to the existing implementations with `JAXB`. However, the drawback of the implementation outside the data binder (`Implementation 2`), which is due to the performance cost, is not reduced here. Therefore, the best choice for `cxf`, in implementation and performance cost, is to choose the implementation inside the data binder by studying `JAXB`.

In the following, we show how our lifting algorithm could be realized in `JAXB` and which are the necessary modifications on the existing default readers for `SOAP` and `RESTful` in `cxf` following `Implementation 1`.

8.1.3.1 An application to `SOAP` using `JAXB`

We start by discussing how `JAXB` can be used to apply the lifting algorithm for the `SOAP` case. We remind here that the algorithm for `SOAP` is a projection without renaming (see the `Lifting Algorithm` for the `SOAP` case in 8.1.2.2).

Based on the `JAXB 2.2` specification [43], the unmarshalling algorithm has the following description for the unexpected elements in the input document:

"if EII. (Element Information Item) property is **null**, then there is no property to hold the value of the element. If validation is on (i.e. Unmarshaller.getSchema() is not **null**), then report a javax.xml.bind.ValidationEvent. Otherwise, this will cause any unknown elements to be ignored.[3] "

Therefore the projection in JAXB is offered for free, when the schema validation is disabled. Now, the question is how to resolve the problem when a schema validation is required.

A prospective solution is to change the service schema such that it will be able to validate documents with unexpected elements. Let us consider for instance the example of interface substitution defined in Figure 8.4. Coming back to our formal type system, the type of the op structure at the server side, we call it t, should be represented as follows in order to ignore all unexpected elements:

$$t = op[\; arg[\; \mu X.(*)[\top], (X + x[int], \top) \;], End \;], End$$

The closest schema definition that matches with t is the following one:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" version="1.0">
  <xs:element name="op" type="op" />
  <xs:complexType name="op">
    <xs:sequence>
      <xs:element name="arg" type="arg"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="arg">
    <xs:sequence>
      <xs:element name="x" type="xs:int"/>
      <xs:any maxOccurs="unbounded"  minOccurs="0"
         processContents="skip"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

[3] See Appendix B.3.4 https://jcp.org/aboutJava/communityprocess/mrel/jsr222/index2.html

We use here <xs:any>[4] which enables any element to appear in the containing sequence. The used processContents is an indicator of how an application or XML processor should handle validation of XML documents against the elements specified by this any element. Using skip value for processContents signifies that the XML processor does not attempt to validate any elements with a namespace other than "x".

In order to generate the previous XSD file using the existing JAXB implementation, the service resource Java code should undergo some changes as in the following. Class Op is defined as follows:

```
@XmlRootElement(name="op")
public class Op {

    @XmlElement(required=true)
    protected A arg;
```

Class A has to add a new attribute any with annotation @XmlAnyElement in order to generate the <xs:any> element in the schema of type "a":

```

public class A {

    protected int x;
    @XmlElement(required = true)
    protected String y;
    @XmlAnyElement
    protected List<Element> any;

    /*Getters and Setters*/
      ...

}
```

However, the required modifications will foul the Java code at the service and the client sides with a useless code: classes have to define an any attribute with type List<Element> with its getter and setter. Another more adequate solution to allow schema validation consists of modifying the existing schema generation and schema compilation functions in JAXB to generate the schema with the <xs:any> type added to complex types without the need to make it explicit at the object level.

Now, the question is:

Does the validation with a schema brings any benefit to an OO framework for Web services?

As the schema in an OO framework is generated from the object level and then used to generate the object level at the client, therefore the message is already well typed at emission at the object level (before marshalling) and there is no need to a schema validation at reception.

[4] http://msdn.microsoft.com/fr-fr/library/ms256043%28v=vs.110%29.aspx

The problem to discuss here is more particularly related to the case when an unsafe user, not properly generated from a schema, sends incorrect messages. We discuss here two cases of message reception, at the server and the client sides:

- *at the server:*
  - if the root of the XML document does not match with any of the service operation names, an exception is thrown before proceeding to unmarshalling. This exception is thrown by the DocLiteralInInterceptor input interceptor in the Unmarshal phase of the cxf inbound chain which validates the input message conformally to the service WSDL file.
  - if the root element is validated by the WSDL, then at unmarshalling, it is possible that the required element for the operation argument is not present in the XML and therefore the operation is invoked with a **null** parameter. In this case, the ServiceInvokerInterceptor in a phase (called Invoke) of the cxf inbound chain, will throw an exception,
  - if the root element is validated by the WSDL and the service operation is invoked with a no-**null** instance, it is possible that some referenced objects in the argument instance are **null** while they are required in the XSD file. Here, it is on the developer to check if the referenced value is not **null** which is a very usual task in OO development practice. Moreover, this is also compatible with the loose coupling principle between the object level and the service level: the Java code should be independent from the schema binding associated to classes.
- *at the client:*
  - if the root of the XML document does not match with the expected return structure, the Unmarshaller instance will throw an exception as it is specified in the JavaDoc of the Unmarshaller interface:

    "An unmarshal method never returns **null**. If the unmarshal process is unable to unmarshal the root of XML content to a JAXB mapped object, a fatal error is reported that terminates processing by throwing JAXBException."[5]
  - if the root element is valid but the XML content does not match with the expected elements, thus a **null** is returned. Here, it is on the developer to check if return value is not **null** which is a usual task in OO development practice.
  - if the root element is valid and the returned value is not **null**, it is possible that some referenced objects in the returned value are **null**. Here is the same case as before.

[5] https://jaxb.java.net/nonav/2.2.4/docs/api/javax/xml/bind/Unmarshaller.html

Therefore, schema validation is not necessary for a safe Web services execution and it could be completely avoided. In Appendix E.1.1, we show how `cxf` can be configured for `SOAP` to apply the lifting algorithm at the server and the client sides.

8.1.3.2 An application to `RESTful` using `JAXB`

We start by discussing how `JAXB` can be used to apply the lifting algorithm for the `RESTful` case. We remind here that the algorithm for `RESTful` is a projection with renaming (see the `Lifting Algorithm` for the `RESTful` case in 8.1.2.2). Based on the `JAXB 2.2` specification[6], the `unmarshal` method could be used in two ways:

- without a declared type in order to return an `Object` instance:

```
 java.lang.Object unmarshal(
  java.io.File arg
or java.net.URL arg
or java.io.InputStream arg
or org.xml.sax.InputSource arg
or org.w3c.dom.Node arg
or javax.xml.transform.Source arg
or javax.xml.stream.XMLStreamReader arg
or javax.xml.stream.XMLEventReader arg
)
```

- with a declared type in order to return a `JAXBElement` which wraps of the unmarshalled object (it is also called unmarshalling as `JAXBElement`):

```
<T> JAXBElement<T> unmarshal(
  org.w3c.dom.Node, Class<T> declaredType
or javax.xml.transform.Source, Class<T> declaredType
or javax.xml.stream.XMLStreamReader, Class<T> declaredType
or javax.xml.stream.XMLEventReader, Class<T> declaredType)
```

Conformally to the unmarshalling algorithm with a declared type, the name of the document root element is not considered. We can find this information more clearly in the `JavaDoc` of the unmarshaller:

> **"Unmarshal by Declared Type:** An unmarshal method with a declaredType parameter enable an application to deserialize a root element of XML data, even when there is no mapping in JAXBContext of the root element's XML name. The unmarshaller unmarshals the root element using the application provided mapping specified as the declaredType parameter.[7] "

[6] See section 4.4 of the `JAXB` specification available at the following address `http://download.oracle.com/otndocs/jcp/jaxb-2.2-mrel2a-oth-JSpec/`

[7] see the appendix B.3.2 from the specification

In other terms, using an unmarshal method with a declared type, the root element of the received document is ignored when an instance of a `JAXBElement` is created. Moreover, the unmarshalling algorithm in `JAXB` allows the projection when the schema validation is disabled, as we discussed for the `SOAP` case. Therefore, again, `JAXB` offers for free the way to apply the lifting algorithm for the `RESTful` case.

As we did for the `SOAP` case, we discuss here the possibility of defining a schema for validating the input document and the utility of this validation. To explain our approach, let us consider the example of value substitution defined in 8.1.1. Coming back to our formal type system, the type of the `A` resource structure, we call it a, should be represented as follows in order to ignore the name of the root element and all unexpected elements:

$$a = (*)[\ \mu X.(*)[\top], (X + x[int], \top)\ ], End$$

Indeed, it is not possible to define a schema to type an unspecified root element. However, the schema could be parametrized using the unmarshalling type and the root name of the input document to validate documents for the lifting algorithm. To explain the idea, we consider the following schema for the value substitution example, which validates an input document with a `B` root element and an additional unexpected element at reception:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" version="1.0">
  <xs:element name="a" type="a"/>
  <xs:element name="b" substitutionGroup="a" type="a"/>
  <xs:complexType name="a">
    <xs:sequence>
      <xs:element name="x" type="xs:int"/>
      <xs:any maxOccurs="unbounded"  minOccurs="0"
         processContents="skip"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

In this schema, the following line:

<xs:element name="b" substitutionGroup="a" type="a"/>

depends on two information known at runtime:

- unmarshalling type, here it is `A` (the input type of `op`), which is used to affect the correct values for the `"substitutionGroup"` and the `"type"` attributes.
- the name of the received root element, here it is `"b"`, which is used to affect the correct value to the `"name"` attribute.

Therefore, it is sufficient to add this line to the schema at runtime in order to accomplish the validation process.

Now, as we did for the SOAP case, we discuss here if the validation following a schema is mandatory in OO frameworks like cxf. As for the SOAP case, in case the incoming message is sent from a safe side, there is no need for a schema validation at reception because the message is type checked at emission. In case the message is sent by an unsafe user, it is possible that the received document does not contain some required elements which will be referenced as **null** in the unmarshalled resource. As we discussed before for the SOAP case, it is on the developer to check if an object reference is **null**.

In Appendix E.1.2, we show how cxf can be configured for RESTful to apply the lifting algorithm at the client and the server sides.

8.2 An adaptation for a loose-coupled schema mapping

In this section, we explain how to configure cxf to avoid the definition of JAXB schema binding at the object level for two reasons:

- In 5.2.3, we showed how schema binding makes the object level tightly coupled to details at the service level,
- In the previous section, we showed that the schema validation is useless for a Web services context when using OO frameworks. Therefore, developers do not need to know all the schema binding complex details to implement Web services.

In the following we discuss three points in order to simplify the mapping schema with the just necessary for an object oriented Web services development: (i) reducing access field complexity, (ii) mapping document root element and (iii) handling object subtyping.

8.2.1 Reducing access field complexity

In 5.2.3, we showed that there are different ways to map the fields in an object type. In order to simplify this complexity, our principle is: conformally to the OO development practice, the developer specifies the access to the class fields through the defined getters and setters. Therefore, the schema generation will produce a default schema type conformally to the specified class field accessors. JAXB handles such a default case without requiring any particular binding schema [8].

8.2.2 Mapping document root element

As we explained before in 5.2.3, the @XmlRootElement is required for RESTful and not for SOAP services. Beside specifying the fields mapping, there is a need in RESTful to map the name of the root document which is done through @XmlRootElement annotation in JAXB.

[8] See Section 8.12.5.1 from the specification

It is ugly to force the use of such an annotation specially that, by default, a document could have the name of its object type. JAXB allows the marshalling and unmarshalling without the need of an @XmlRootElement annotation on the marshalling class, using JAXBElement[9].

In order to avoid the mandatory use of this annotation, the reader in the Marshal phase and the writer in the Unmarshal phase should be configured in order to call the data binder using JAXBElement for marshalling and unmarshalling. Appendix E.2 presents more details about the cxf configuration in RESTful in order to resolve this problem.

8.2.3 Handling object subtyping

In 5.2.3, we have showed that in order to handle subtyping an annotation is required, @XmlSeeAlso. In the following, we discuss the two subtyping cases in an OO language, inheritance between classes and use of interfaces, in order to make subtyping more natural conformally the the OO development practices without the need of any additional configuration.

- inheritance between classes: When a service defines an operation $op : A \rightarrow void$, which can accepts subclasses instances of B (subclass of A), the developer has to annotate the A class with @XmlSeeAlso{B.**class**} annotation. The role of the @XmlSeeAlso annotation is to load additional classes (here it is class B) to the JAXB context when the class, where this annotation is defined (here it is class A), is loaded into the context.

  In cxf, the JAXB context is created each time a service operation is invoked. For instance, when a client uses a B instance to call op, the writer creates an instance of the JAXB context using the dynamic type B (as in cxf 2.5.x implementation) or the static type A (as in cxf 2.7.x implementation). When the dynamic type is used, there is not a need to add the @XmlSeeAlso annotation on class A, because B is directly loaded into the context. When the static type is used, class A is loaded into the context. Therefore in order to load also the subclass B to the created context, the use of @XmlSeeAlso annotation is required.

  In the same manner, at unmarshalling, loading the subclasses in the context allows the creation of subtype instances instead of supertype ones. When a document is marshalled using a subtype where a supertype is intended, the marshalled document contains a parameter referring to the used subtype. For instance in case an instance of a B class is marshalled using a declared type A, supertype of B (using for instance:

  ```
  <T> JAXBElement<T> unmarshal(javax.xml.stream.XMLStreamReader , Class<
                                  T> declaredType)
  ```

  ), while B is known in the JAXB context, the root element of the corresponding document will have the map name of class A, let us say a, and a parameter referring to the subtype b and the structure of the b type, as defined in the following:

[9] See Appendix B.3.3 for unmarshalling and B.4.2.1 for marshalling from the specification

```
<a xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
   type="b">
  <!--Here are the elements of the b type-->
  <x>x0</x>
  <y>y0</y>
</a>
```

Therefore, in order to avoid the forcing of using of `@XmlSeeAlso` annotation, we propose that the `JAXB` context must contain, at initialization, all classes with their subclasses as defined in the `Java` code package. Using all defined classes, the `JAXB` instance will be the same for all the service invocations. Therefore, it is useless to create a new `JAXB` context each time an operation is invoked at marshalling and unmarshalling. The adequate solution is to create a single `JAXB` instance to be used by the reader and the writer.

In Appendix E.3, we show how to configure `cxf`, in order to create a single and global `JAXB` instance for `RESTful` and `SOAP` services.

- use of interfaces: When a service defines an operation, let us call it *op* such that $op : I \rightarrow void$ with I is an interface having two implementation classes (`A` and `B`), the developer has to:
  - add `@XmlSeeAlso({A.class, B.class})` on the `I` interface in order to load `A` and `B` classes to the `JAXB` context,
  - add `@XmlJavaTypeAdapter(AnyTypeAdapter.class)` annotation to `op`. This annotation uses the class `AnyTypeAdapter` defined as follows:

    ```
    public class AnyTypeAdapter extends XmlAdapter<Object,
       Object> {
      Object unmarshal(Object v) { return v; }
      Object marshal(Object v) { return v; }
    }
    ```

    At the creation of a `JAXB` instance, classes loaded into the context should be marshallable types (define a default constructor and getter/setter methods on fields). When interfaces are used as input or output type of a service operation, they can not be loaded into the context because they does not satisfy the conditions of a marshallable type. In order to resolve this problem, `JAXB` proposes the use of the general adapter, `AnyTypeAdapter`, which adapts the use of `I` to `Object` class which satisfies the `JAXB` conditions. Then at marshalling/unmarshalling, the objets/documents will be marshalled/unmarshalled using the `A` or `B` classes loaded into the context. However, `A` and `B` classes could be also not marshallable types. For instance, the developer may require immutable objects as input on `op`. Thus the interface `I` and its implemented classes define only getters without setters on the required fields. Therefore, in order to give the developer a complete liberty on OO development without

requiring any particular constraint, in the following we propose to define a more specific adapter for interface `I` by avoiding the use of the `@XmlSeeAlso` annotation.

To solve this problem, we have to define an `adapter` class, that extends `XmlAdapter`[10]. This adapter will convert the instances of type `I` to a marshallable class that `JAXB` knows how to handle.

This marshallable class, let us call it `AdaptedI` class, defines only the getter methods declared in the interface and their corresponding fields. Then for each field, it must define a setter method.

The `adapter` should define methods for adapting a bound type to a value type or vice versa. In other terms that allows to convert from an instance of type `I` into an instance of `AdaptedI` and vice versa. The `adapter` methods are invoked by the `JAXB` binding framework during marshalling and unmarshalling:

- `AdaptedI marshal(I i)`: During marshalling, `JAXB` binding framework invokes `XmlAdapter.marshal(..)` to adapt a bound type (an instance of type `I`) to value type (an instance of `AdaptedI`), which is then marshalled to `XML` representation.
- `I unmarshal(AdaptedI ai)`: During unmarshalling, `JAXB` binding framework first unmarshals `XML` representation to a value type (an instance of `AdaptedI`) and then invokes `XmlAdapter.unmarshal(..)` to adapt the value type to a bound type (an instance of type `I`).

Let us consider that the `I` interface is defined as follows:

```
public interface I{

public X getX();
public Y getY();
public String toString();
}
```

The `AdaptedI` class should be defined as follows:

```
public class AdaptedI{

private X x;
private Y y;

public X getX();
public void setX(X x);

public Y getY();
public void setY(Y y);
```

[10] http://docs.oracle.com/javase/7/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html

```
}
```

The `Adapter` class is defined as follows:

```
public class Adapter extends XmlAdapter<AdaptedI,I>{

  @Override
  public AdaptedI marshal(I arg0) throws Exception {
    AdaptedI i = new AdaptedI();
    u.setX(arg0.getX());
    i.setY(arg0.getY());
    return i;
  }

  @Override
  public I unmarshal(AdaptedI arg0) throws Exception {

      I i = new I() {

            private X x;
    private Y y;

     public X getX(){
       return this.x;
     }
    public void setX(X x){
      this.x=x;
    }

     public Y getY(){
      return this.y;
     }
    public void setY(Y y){
      this.y=y;
    }
      };
      i.setX(arg0.getX());
      i.setY(arg0.getY());
      return i;
  }
}
```

In the `marshal` method, the creation of an `AdaptedI` instance from an instance of type `I` is easy using the getters and setters. The `unmarshal` method is more delicate. This method requires returning a instance of type `I` from the received `AdaptedI` instance. However, multiple implementation classes could exist for `I`, so the question is: which

implementation class to choose for creating an instance of type `I`? We can resolve this issue using an anonymous class which allows the factory of an instance of type `I`.

In order to enable the use of the new adapter, the annotation `@XmlJavaTypeAdapter(Adapter.class)` should be added to the input or output parameter of the service operation.

8.3 Merging all required adaptations

After presenting separately how to resolve the interoperability problems and schema binding loose coupling problems, the question now is:

Is it possible to merge all the proposed solutions in one configuration which could be then standardized for the `cxf` framework?

In this section, we show first that we can deduce a standard configuration without having any contradiction between the proposed solutions in 8.1 and in 8.2. Then, we present an automation of this configuration at the development of the service and the client. We finish by presenting a performance study of our solution.

8.3.1 A standard configuration

Table 8.1 presents a summary of the previous presented solutions for `SOAP` and `RESTful` services in `cxf` framework. Based on this table, we present in Appendix E.4.1 a standard `cxf` configuration that fixes all the previous discussed problems.

8.3.2 Automation

In order to automate the `cxf` configuration defined in Appendix E.4.1, we propose the following facilities at the client and the server sides:

- at the server side: we provide a `deploy` algorithm which should be executed at the deployment of the `Java` code as a service (static time configuration). The `deploy` algorithm checks first for interface types in the parameters or the return type of the service methods in order to generate the corresponding adapters. Then, it applies the necessary configuration to build a single and global `JAXB` context at execution. Finally, a `cxf` configuration file is created based on the input parameters of `deploy` and the `JAXB` context configuration. This file differs between `SOAP` and `RESTful` services by applying the necessary modifications as described in Table 8.1.

- at the client side: we propose a new API which provides a `getProxy` operation. `getProxy` gets an OO service required interface and a service `URL` address to return a corresponding proxy. The `getProxy` declaration is so defined as in the following:

```
<T> T getProxy(Class<T> c, String url)
```

Substitution principle	Loose coupled schema binding			
	Reducing access field complexity	Avoiding @XmlRootElement	Handling object subtyping	
			Inheritance between classes	Using interfaces
SOAP	Disable schema validation Using the field accessor methods without annotations	Nothing to do	Define a global JAXB context with all the package classes	Using an interface adapter with anonymous class
RESTful	Unmarshal as JAXBElement Using the field accessor methods without annotations	Marshal and Unmarshal as JAXBElement	Use a single JAXB context with all the package classes	Using an interface adapter with anonymous class

Table 8.1: A summary of our proposed solution to fix interoperability and loose coupling problems in cxf

It returns a proxy of type T, the type of the required interface given in the first parameter. This returned proxy is built from a generated cxf configuration file. Building this configuration file is very similar to the building process of the configuration file at the server.

More details about the deploy and getProxy algorithms in SOAP and RESTful are presented in Appendix E.4.2.

8.3.3 Performance study

In order to evaluate the overhead of using our lifting algorithm for SOAP and RESTful models, we use MBeans to compare the execution of a service request before and after using our algorithm by considering some performance metrics. In the following, we introduce briefly MBeans and its integration to cxf, our test cases and we finish by an analysis of the test results.

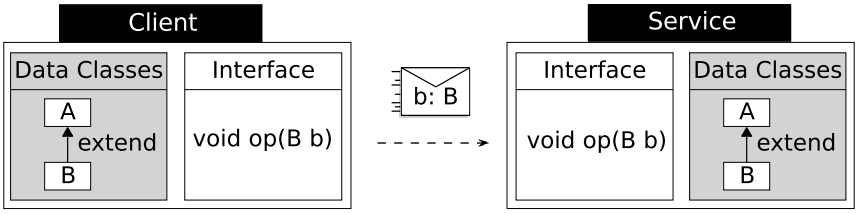
8.3.3.1 MBeans and cxf

MBeans, or (managed beans), is a managed Java object, similar to a JavaBeans component, that follows the design patterns in the JMX (Java Management Extensions Instrumentation) specification [79]. An MBean can represent a device, an application, or any resource that needs to be managed. MBeans exposes a management interface that consists of the following: a set of readable or writable attributes, or both, a set of invokable operations and a self-description[11]. In Appendix E.5, we show how to configure cxf to get the MBeans statistics at the service execution.

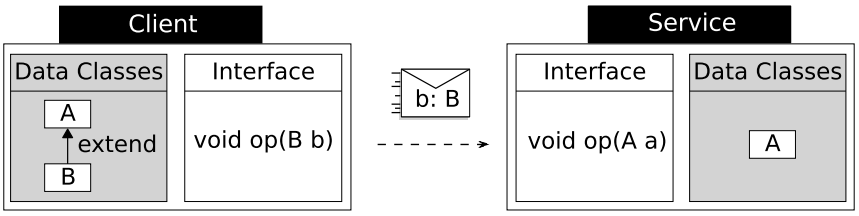
In order to see the MBeans of an execution on a server, we need a JMX monitoring software. We choose JConsole[12] from the JDK. JConsole uses the extensive instrumentation of the Java Virtual Machine to provide graphical information about the performance and resource consumption of applications running on the Java platform.

[11] http://docs.oracle.com/javase/tutorial/jmx/mbeans/

[12] http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html



(a) Test case for the standard `cxf` configuration



(b) Test case for our `cxf` configuration verifying the lifting algorithm

Figure 8.10: Test cases

8.3.3.2 Test cases

In order to evaluate the overhead of our lifting algorithm, we consider to compare at the server: (i) the unmarshalling of a document of type `B` subtype of `A` when `A` and `B` are known at reception, using the standard `cxf` configuration (see Figure 8.10(a)), (ii) with the unmarshalling of this same subtype document when `B` is unknown at reception, using our proposed `cxf` configuration to apply the lifting algorithm (see Figure 8.10(b)).

In order to have correct test measurement, we consider the following requirements:

- the used data for unmarshalling must be tough in width and depth, in order to amplify the projection cost. We choose a simple structure of type `A`, and a complex structure of its subtype `B` as represented in the UML diagram of Figure 8.11, (we suppose that only `A` and `X` classes are known at reception for the test of Figure 8.10(b)).

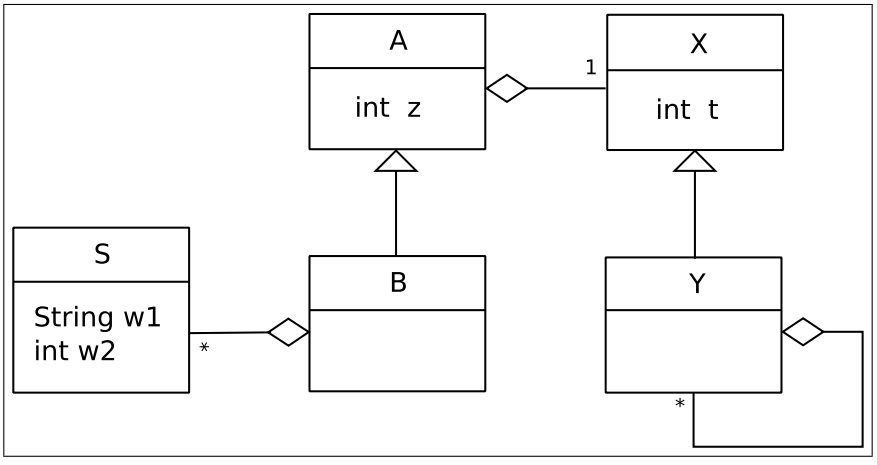


Figure 8.11: UML diagram for A and B details of classes structure

- the performance metrics measured must be an average of multiple requests execution at the server in order to minimize the cost of hardware and system cost.

Therefore, we choose to make requests on the server using a `B` instance as follows:

```
        B b = new B();
        b.setZ(2);

    // extending b in width
        ArrayList<S> list = new ArrayList<S>();
        for(int i=0; i< 48000; i++){
          S s = new S();
          s.setW1("test");
          s.setW2(22);

          list.add(s);
        }
                    b.setListS(list);

                    // extending b in depth
        Y y1 = new Y();
        for(int i=0; i<92; i++){
          Y y2= new Y(y1);
          y1=y2;
        }

        Y y = new Y(y1);
        y.setT(18);
        b.setX(y);
```

The algorithm must project the corresponding marshalled structural document of this `B` instance in order to get at reception by unmarshalling what corresponds to the following `A` instance:

```
    A a = new A();
    a.setA1(2);

    X x= new X();
    x.setT(18);
    a.setX(x);
```

Figure 8.12 gives an idea how important is this projection.

8.3.3.3 Results

In the following, we present the result of our test for `SOAP` and `RESTful` models. These tests are done on Mac OS X (version 10.6.8) using `Java` SE 6, Tomcat v7.0.52 and `cxf` v2.7.10. Tests are executed at localhost using a loop of client requests.

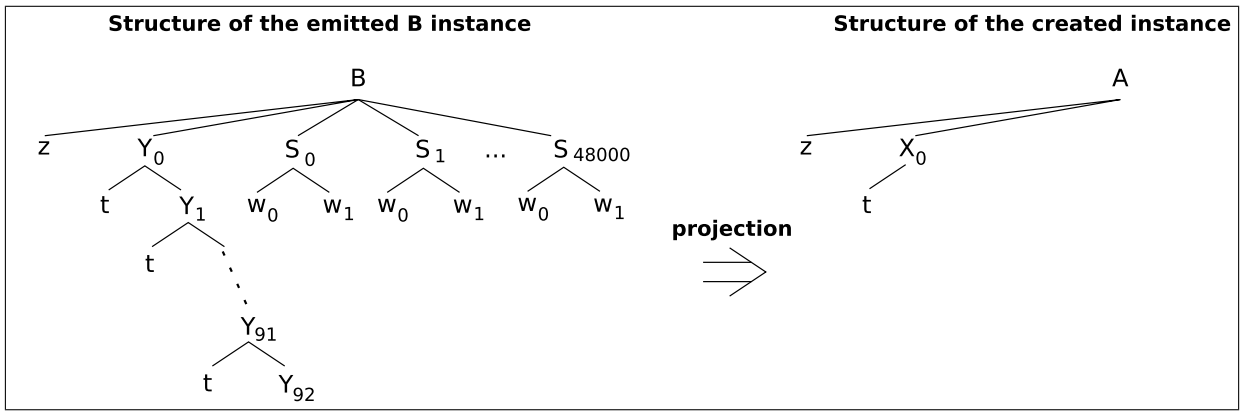


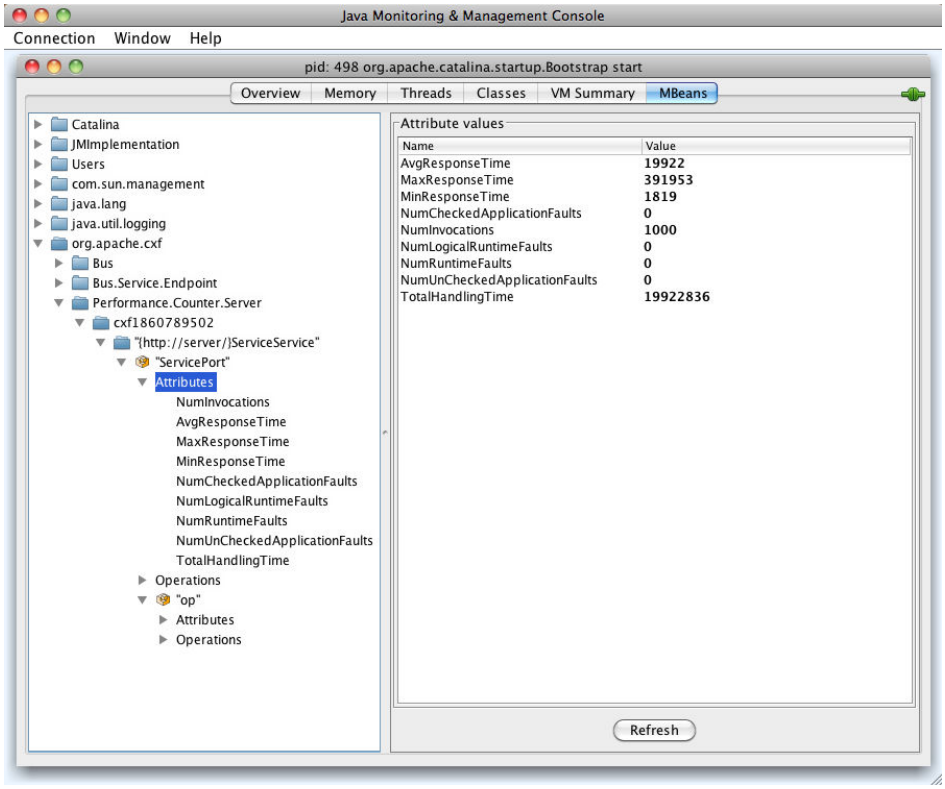
Figure 8.12: The structure of the emitted B instance and its corresponding creates an A instance by projection

We refer in the following to the results of the execution of Figure 8.10(a) test case by *"Before"* and to the results of the execution of Figure 8.10(b) test case by *"After"*.

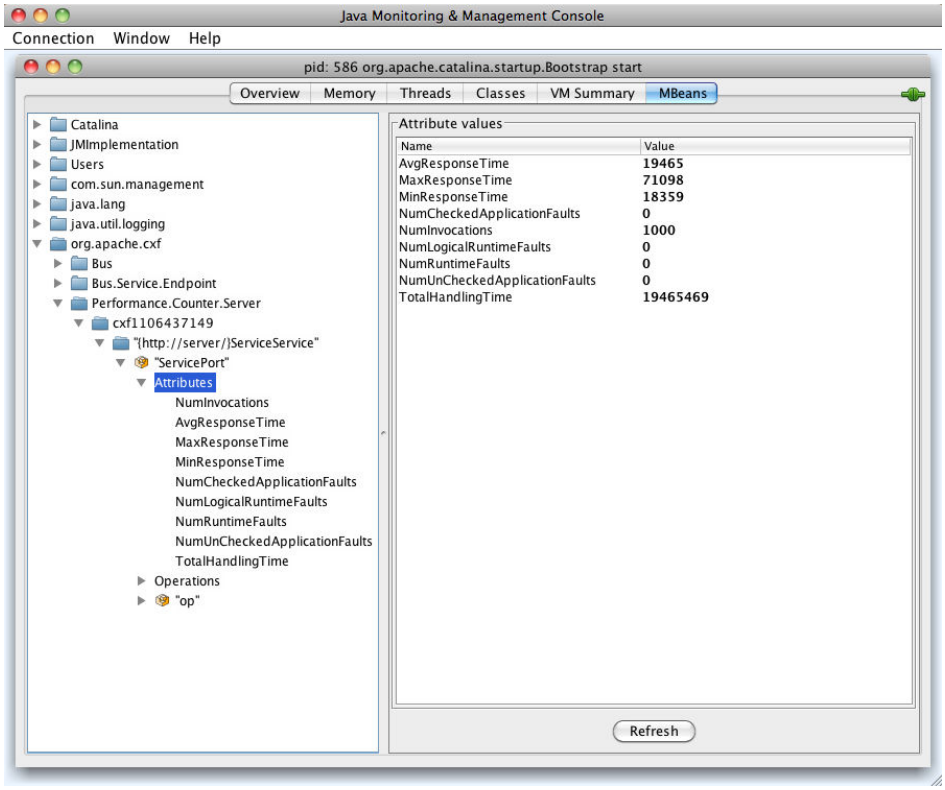
SOAP.

1. `MBeans` statistic results are represented in Figure 8.13.

2. an execution overview is represented in Figure 8.14.

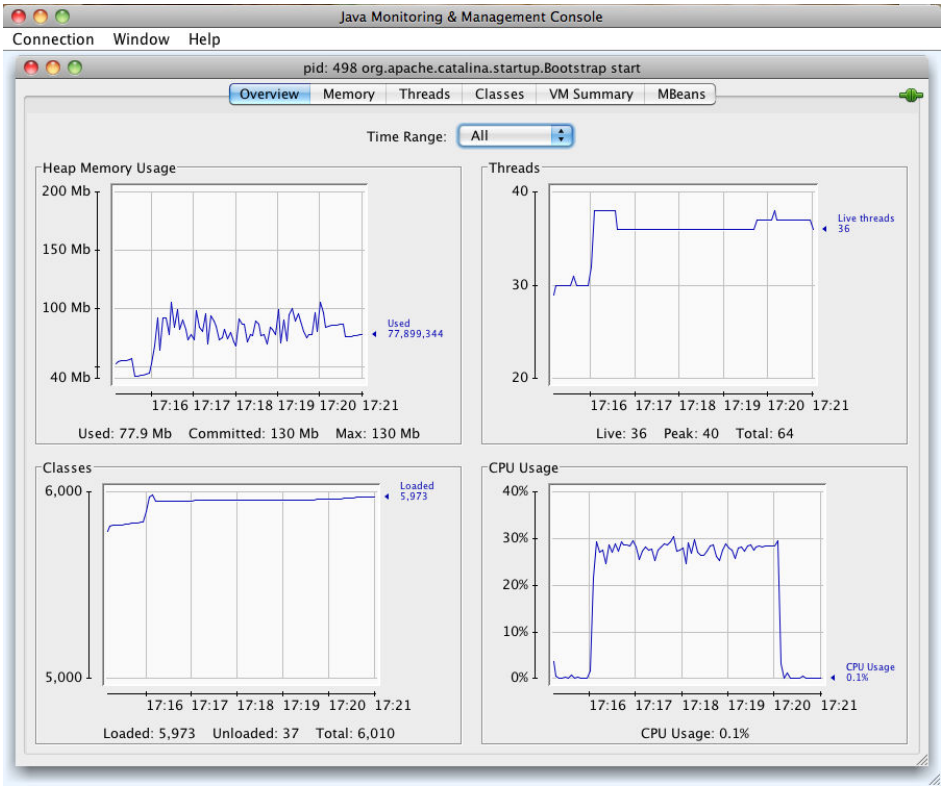


(a) Before

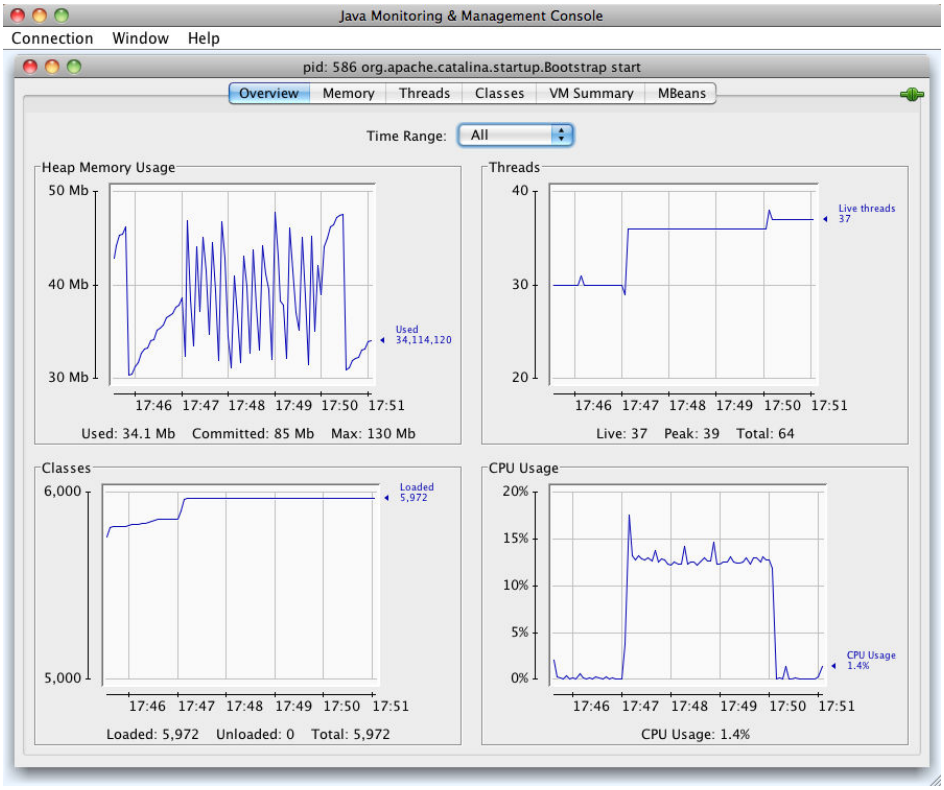


(b) After

Figure 8.13: `MBeans` statistic results for `SOAP`



(a) Before



(b) After

Figure 8.14: Execution overview for SOAP

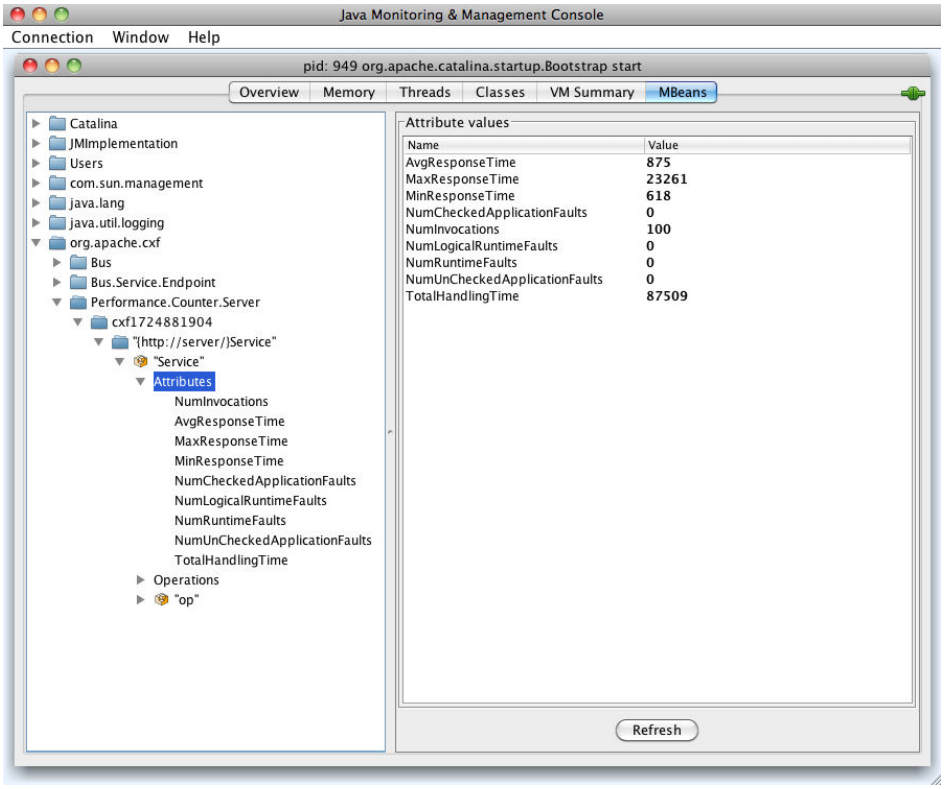
RESTful.

1. `MBeans` statistic results:

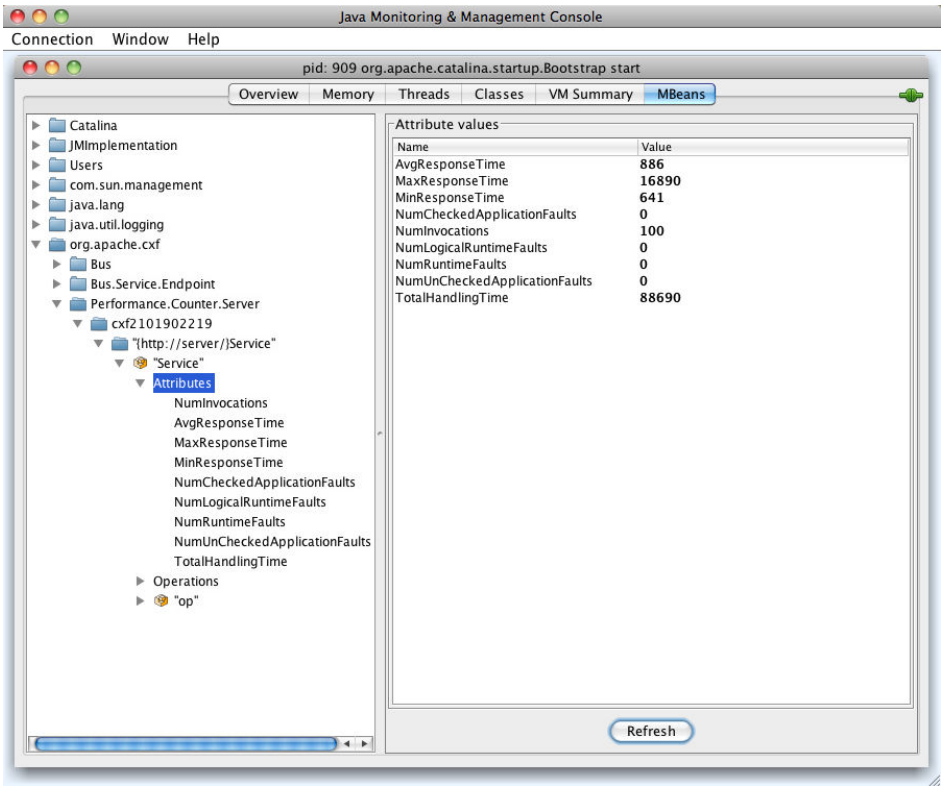
    - using `JSON`, are represented in Figure 8.15,
    - using `XML`, are represented in Figure 8.16.

2. an execution overview :

    - using `JSON`, is represented in Figure 8.17.
    - using `XML`, is represented in Figure 8.18.

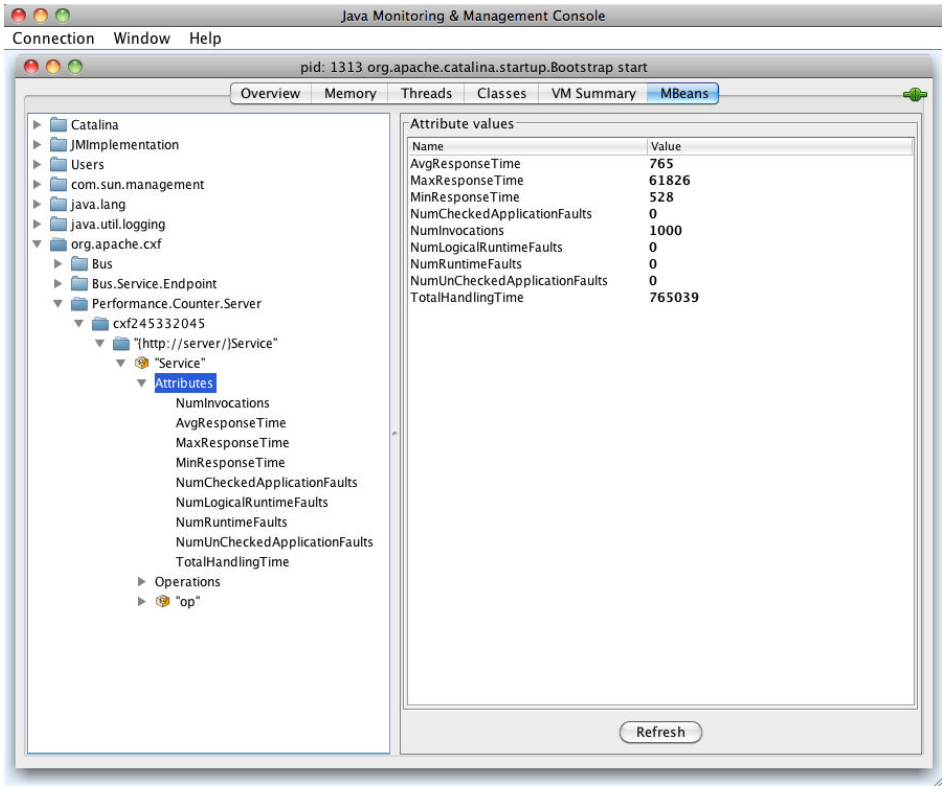


(a) Before

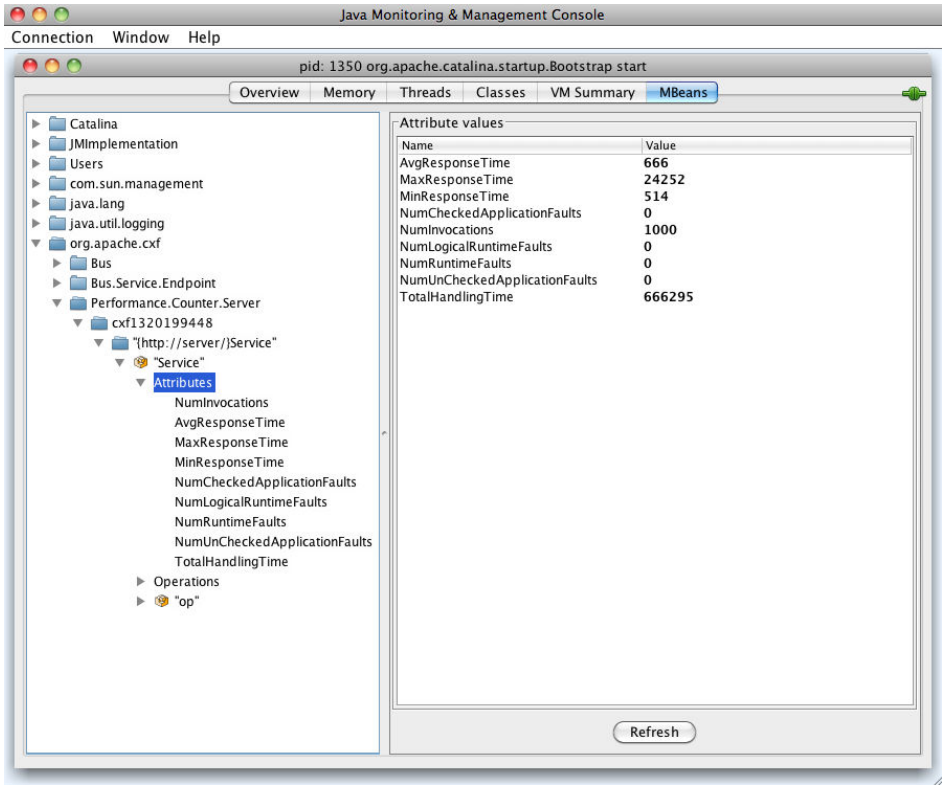


(b) After

Figure 8.15: `MBeans` statistic results for `RESTful` using `JSON`



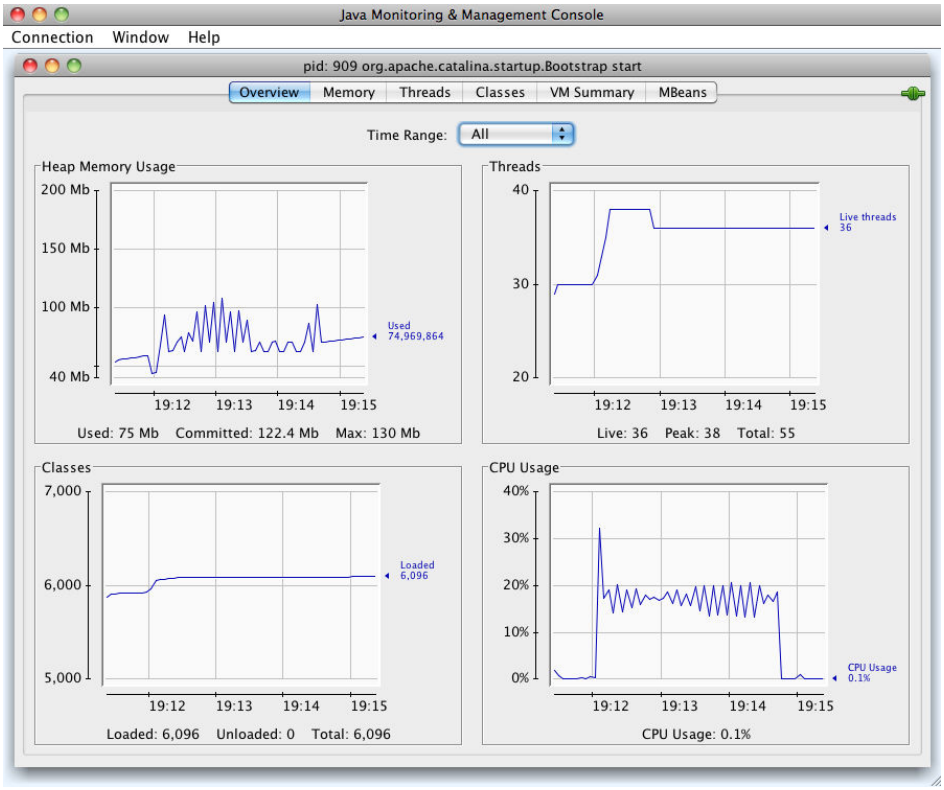
(a) Before



(b) After

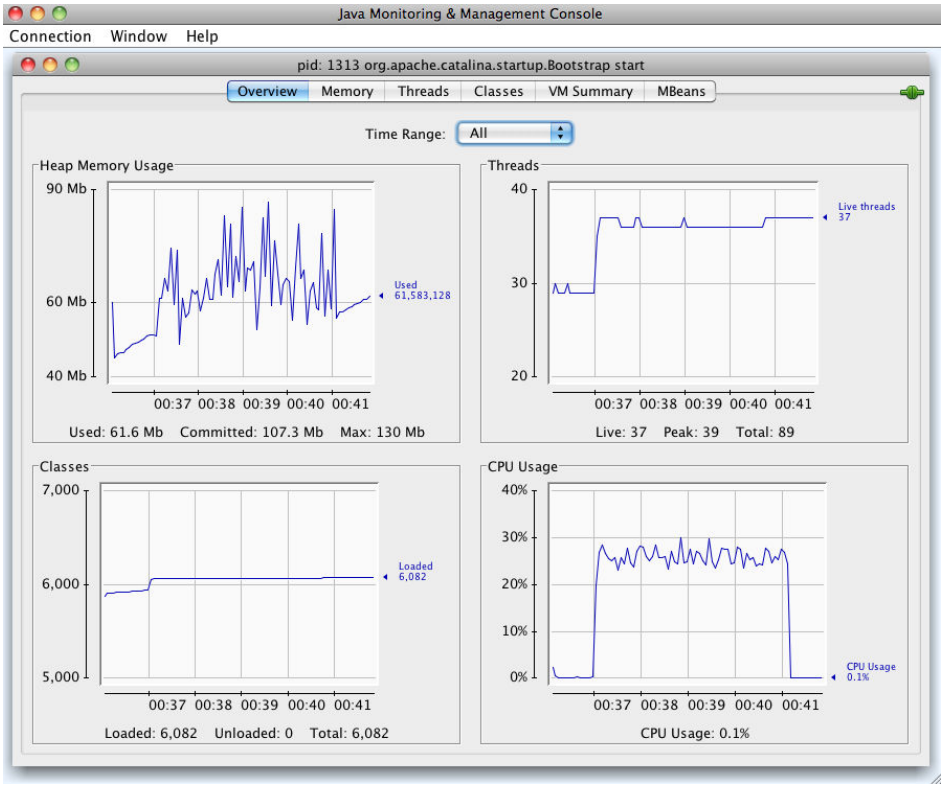
Figure 8.16: `MBeans` statistic results for `RESTful` using `XML`

(a) Before

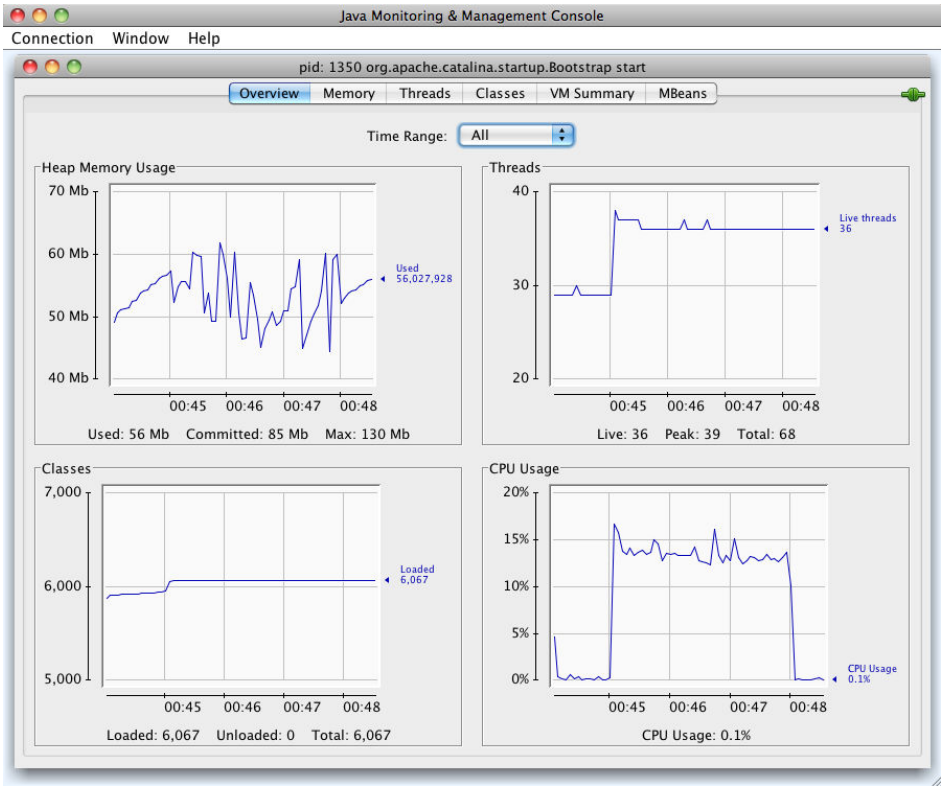


(b) After

Figure 8.17: Execution overview for `RESTful` using `JSON`



(a) Before



(b) After

Figure 8.18: Execution overview for `RESTful` using `XML`

8.3.3.4 Discussion

The previous presented figures for test results show that the application of our algorithm has no impact on runtime performance (slightly better).

Results on SOAP show that the performance is better when using our algorithm by considering the following measures:

- the average response time is 19922 ms (after) compared to 19465 ms (before) (see avgResponceTime parameter in Figure 8.13),
- the total time to respond 1000 client requests is around 3 minutes (after) compared to 4 minutes (before) (see CPU Usage graph of Figure 8.14),
- the CPU usage is around 13% (After) compared to 30% (before) (see CPU Usage graph of Figure 8.14),
- the Heap memory usage changes in an interval between 30 Mb and 50 Mb (after) compared to an interval between 50 Mb and 100 Mb (before).

These results can be explained by the fact that the projection offered by JAXB (which we used for our lifting algorithm) does not travel all the document tree but select only the needed elements from this tree according to their tags. That costs less time and runtime object creation which explains the decrease in the average response time, the total response time, the CPU usage and the Heap memory usage.

For RESTful, we distinguish two cases depending on the format of the treated data: JSON or XML:

- JSON case:
  - the average response time is very similar (before and after) (see Figure 8.15),
  - the total time to respond 100 client requests is less than 3 minutes (after) compared to more that 3 minutes (before) (see the CPU Usage graph in Figure 8.17),
  - the CPU usage is almost similar (before and after) (see the CPU Usage graph in Figure 8.17),
  - the heap memory usage is also very similar (before and after) (see the Heap Memory Usage graph in Figure 8.17)
- XML case:
  - the average response time is 666 ms (after) compared to 765 ms (before) (see avgResponceTime parameter in Figure 8.16),
  - the total time to respond 1000 client requests is around 3 minutes (after) compared to 4 minutes (before) (see CPU Usage graph of Figure 8.18),

- the CPU usage is around 13% (after) compared to 30% (before) (see CPU Usage graph of Figure 8.18),
- the Heap memory usage changes in an interval between 40 Mb and 60 Mb (after) compared to an interval between 50 Mb and 90 Mb (before) (see the Heap Memory Usage graph in Figure 8.18).

The reason for the slight differences between results after and before is as we explained previousely for the SOAP case. The differences between results using JSON or XML is due to the fact that JSON is lighter than XML, which costs less time to process data. That explains why we can not see a difference for the average response time and the heap memory usage for the JSON case.

In conclusion, a client can switch dynamically from one server treating subtypes to another which does not by guaranteeing that at least the server performance is not decreased. Indeed, the complexity remains linear over the size of the structure (document).

8.4 Conclusion

The chapter shows how to recover the substitution principle and loose-coupling in an OO framework for web services, namely cxf.

First, we have proposed a specification of the data binding used to relate data and types between the object level and service level. 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 specify 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 solid foundations, basic category theory, the specification can be fully formalized. Thanks to the intuitive interpretation of this specification, two implementations were discussed: (i) by modifying the data binder or (ii) by modifying the OO framework. We have deduced that the first implementation is less costly for cxf using JAXB data binding. We have presented how cxf should be configured, in SOAP and RESTful, in order to verify the specification.

Second, we have explained how to configure cxf to avoid the definition of JAXB schema binding at the object level. We have treated three points: (i) reducing access field complexity, (ii) mapping document root element and (iii) handling object subtyping.

Third, we have presented an overview of the cxf configuration, which merges the proposed configurations in respect with the substitution principle and loose-coupled schema binding. We proposed an automation algorithm which generates the required configurations at the client and the server sides for SOAP and RESTful. In order to evaluate the overhead of our proposed configuration to cxf, we have presented a performance study using MBeans. We have proved that at least our proposed solution does not have bad effects on the service performance.

IV

Perspectives

9

Conclusion

In the following, we present an overview about the contributions presented in this thesis.

9.1 Weaknesses in existing OO frameworks

In this thesis, we have presented the weaknesses in the existing OO frameworks for Web services in respect with `loose coupling` and `substitution principle`. We have mainly detailed three problems:

- Tight coupling between the discovery querying at the object level and the used discovery model at the structural level: following three examples using `Java` APIs for `UDDI`/`WS-Discovery` standards (for `SOAP` discovery) and `Linked data` (for `RESTful` discovery), we have proved that the used discovery protocol at the service level is not transparent for development at the object level,

- Weak interoperability in respect with the substitution principle: following two examples applied on the `cxf` framework for `SOAP` and `RESTful` models and which illustrate the application of the substitution principle (on values and interfaces), we proved that there is a lack of interoperability and a strong coupling between clients and servers when OO subtyping is used.

- Tight coupling in binding schema: following some examples on `JAXB` data binding, we have showed that the mapping between object types and their corresponding schema tightly depends on the Web services technology and does not fit with modular OO development practices (subtyping and updates).

9.2 A unified model for Web services

The previous presented problems are mainly due to the diversity of existing implementation techniques of Web services at the service level and which has bad effects at the object level. We have discussed how these problems could be resolved if these Web services technologies could be unified on common concepts and under a whole abstract behavior. Therefore, we have introduced a high-level model based on a chemical semantics for service interactions and dynamic service discovery with first-class channels. Our proposed model is based on few concepts. Web services are viewed as abstract agents exchanging messages via the network. Services are available thanks to the notion of communication channels. Messages can carry channels, thus ensuring full channel mobility. This formal model supports a sound type system. The resulting system supports contravariant types for channels, type checking and type inference. We have motivated by examples the utility of our expressive type system for Web services. Furthermore, the type system accommodates subtyping and general set-based type operators. This generality has been achieved by applying the principles of semantic typing to the Web services world. Moreover, we have shown fundamental correctness properties of the type system in the context of malicious agents and insecure channels.

9.3 A unified object-oriented API for dynamic discovery

Based on our unified model, we presented a new OO API for Web services discovery that hides from technical details at the service level. To define such an API, we have first showed how the details of the standard interfaces (`WSDL` and `WADL`) could be simplified and abstracted. Then, we have deduced a general abstraction of the existing dynamic discovery protocols conformally to our formal model. We have showed how this unified API could improve the existing discovery APIs for `RESTful`/`SOAP` services. Another neutral implementation was presented and which works for whatever Web services model. This implementation is based on the principle of using type structures to discover new services. In the client request for the registry, the message contains an interface representation based on its type. For that, we have used the unified formalization of interfaces in our model. This new implementation brings a big advantage compared to the exiting ones: using of subtyping in the discovery mechanism.

9.4 A new specification for data binding

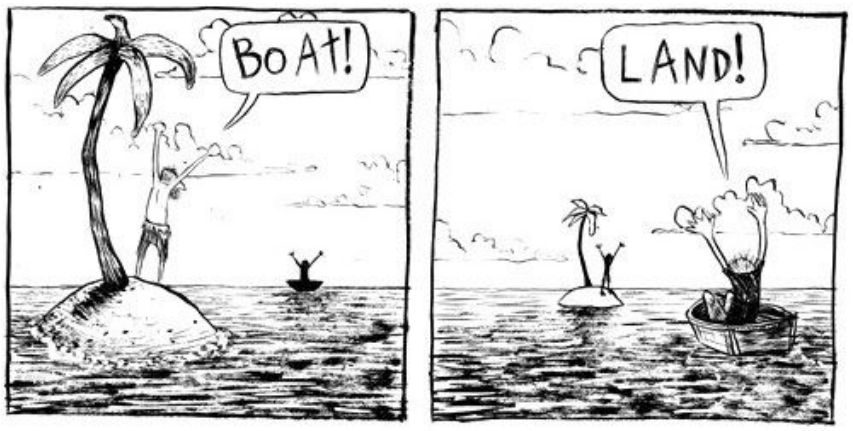
In this thesis, we have proposed a specification of the data binding used to relate data and types between the object and service levels. The specification is splitted into two subsets, the core requirements, which are generally implicit but satisfied by current data bindings, especially by `JAXB`, the default one for `cxf`, and the new requirements, dealing with subtyping and allowing the substitution principle to be recovered. The new requirements mainly specify 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 solid foundations, basic category theory, the specification can be fully formalized.

Thanks to the intuitive interpretation of this specification, two implementations were discussed: (i) by modifying the data binder or (ii) by modifying the OO framework. We have deduced that the first implementation is less costly for `cxf` using `JAXB` data binding. We have presented how `cxf` should be configured, in `SOAP` and `RESTful`, in order to verify the specification. Furthermore, we have explained how to configure `cxf` to avoid the definition of `JAXB` schema binding at the object level. We have treated three points: (i) reducing access field complexity, (ii) mapping document root element and (iii) handling object subtyping. Moreover, we have presented an overview of the `cxf` configuration, which merges the proposed configurations in respect with the substitution principle and loose-coupled schema binding. We have proposed an automation algorithm which generates the required configurations at the client and the server sides for `SOAP` and `RESTful`. In order to evaluate the overhead of our proposed configuration to `cxf`, we have presented a performance study using `MBeans`. We have proved that at least our proposed solution does not have bad effects on the service performance.

10

Future Work



[1]

On one side, we have defined a formal model with an expressive type system. We used this model to unify concepts at the service level in order to make discovery technical details transparent at the object level in an OO framework for Web services. Our model could be useful for other uses and applications as we will briefly discuss in the following. We present in Section 10.1 some perspectives to improve the registries implementation and protocol using this model. Moreover, in Section 10.2 we show how this model can be useful to unify applying security policies for `SOAP` and `RESTful` models.

On other side, we discussed problems in a distributed OO Web services environment concerning: Interoperability by subtyping (substitution principle) and loose coupling. Other problems exist in these environments and are not discussed in this thesis. We present in this chapter five perspectives in this context: (i) In Section 10.3, we present a problem in schema generation when generic types are used in an object type to be converted into a structural type. (ii) In Sec-

[1] http://spiritualartwork.wordpress.com/2013/05/06/its-all-about-perspective-humor/

tion 10.4, we propose to improve RMI and CORBA in order to resolve the problems discussed in this thesis when a structural layer is required for these two technologies. (iii) In Section 10.5, we discuss what is required to make Web services a real OO distributed environment. (iv) In Section 10.6, we propose in future work to implement our proposed specification in other frameworks, (v) Finally, in Section 10.7, we throw a new problematic around matching the nominal type system of the object level with the structural type system of the service level.

10.1 Improvements in dynamic discovery methodologies

We discuss in the following two perspectives for the two dynamic discovery methodologies: (i) Updating service access methodology which requires modifying the service access information each time a switch is done while keeping the required interface unchanged. (ii) Interface generation methodology which needs the generation of the required interface each time a switch is done.

Updating service access methodology. In Chapter 7, we have presented a new API for dynamic discovery based on interface subtyping. We have presented how the API should be implemented for use at the client side. Future work can go more in details about the implementation of this API at the registry side by considering two points:

- Subtyping: The implemented code at the registry side must consider the subtyping rules as defined by Castagna in [29]. We note here that in order to ensure a correct discovery mechanism, a semantical analysis must be applied between the required interface and the provided one. Indeed, two interfaces could be linked with a subtype relation, however, they are dedicated for different uses. Optimization questions could rise here in order to optimize the search time specially when the number of services declared at the registry is huge.

- QoS: Our formalization of dynamic discovery, as presented in Chapter 7, considers that the registry returns one service reference (channel) to be discovered as a required service at the client interface. This service should be selected from a list of candidate services. In this thesis, we were completely transparent on how the service is selected from this list. Future work, could add some QoS requirements at the registry which automatically returns one selected service location based on some criterions.

Interface generation methodology. Two Web services could be semantically equal without providing the same (or subtype) interface. The difference between these two interfaces could be due to: (i) the data type, for instance security requirements could affect the structure and the form of exchanged data, (ii) multiple interactions, a service could be achieved in just one call/return or in multiple successive interactions, like for OAUTH, in order to get the same reply. Thanks to the expressivity of our type system, these requirements are easily represented as we have showed in Chapter 6. In other terms, the type of a service could give a global idea about the

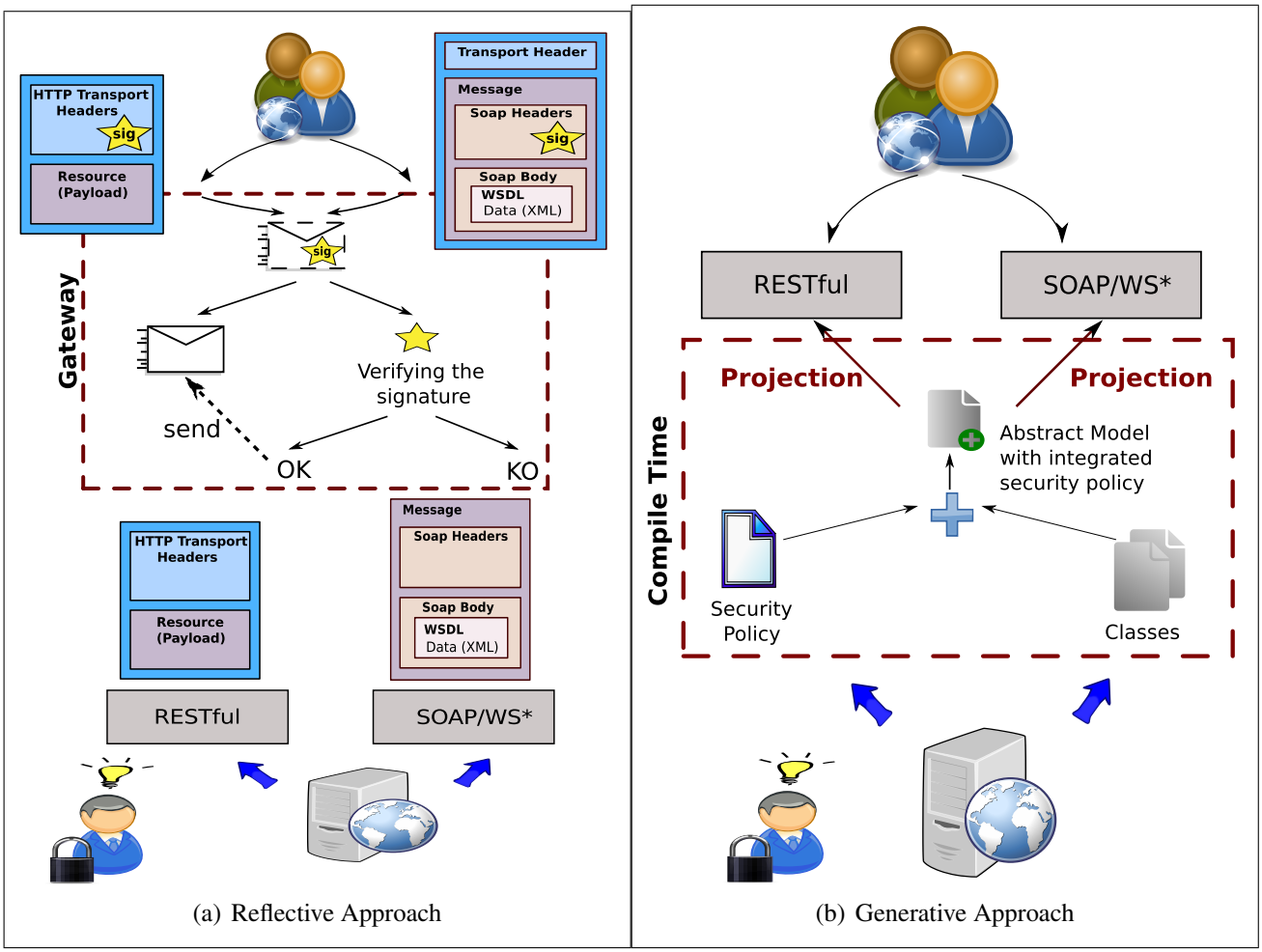


Figure 10.1: Applying an authentication policy at dynamic and static time on a Web service exposed with both models: `RESTful` and `SOAP`

QoS of the service: the required exchanged data, the interaction complexity, etc. Future work could then consider complex types as a QoS criterion for services.

10.2 A new concept-oriented security language

The diversity in Web services provides only very few guarantees concerning security and safety properties of the resulting software systems. Furthermore, different implementations of the same security policy exist. Thus, weaknesses appear in security policies because they are tightly coupled to the Web services implementations. Hence, there is a need for a concept-oriented security policy in order to separate technical details from policy concepts.

In future work, it will be useful to define a concept-oriented security language based on the unified concepts in our formal model. Some security properties like confidentiality, integrity and authentication could be considered. This security language could be applied at run time or

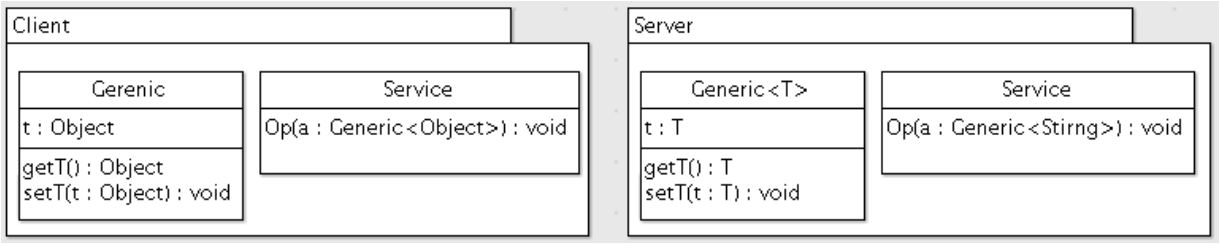


Figure 10.2: Client Code Generation problem for a client interface while using a generic type

at static time on the concerned Web service. For that, we distinguish between two approaches:

- Reflective approach: it consists of separating the security treatment from the Web service implementation. It is a run-time verification approach.
- Generative approach: it consists of enforcing the security policy with the Web service implementation at compile time. It is a static verification approach.

To illustrate the application of these two approaches, Figure 10.1 clearly presents an example of an authentication policy. We consider a server which exposes its service using both `SOAP` and `RESTful` models. By using the reflective approach, the authentication policy must be expressed using the unified model in a gateway intercepting messages before they get the `SOAP` or `RESTful` interfaces. At run-time, either if the intercepted message is designated to the `SOAP` service or to the `RESTful` service, it will be represented with a unified message form in the gateway to submit the same treatment over an algorithm for security checks. While by using the generative approach, the security policy based on concepts will be defined in a separated file from the implementation code (written in `Java` for instance). Then by combining at compile time this security policy with the Web service code, we get a correspondent representation in our abstract model with the integrated security policy. Hence, by a simple projection from this formal representation, we get a secure implementation for `SOAP` or `RESTful` models.

Moreover, our unified model and the type system could be useful to solve some existing problems like security for OAUTH protocol [7, 62, 73].

10.3 Having a full correct schema generation

In this thesis, we have supposed that our object model is correctly represented in the structural model. However, this is correct for cases where there is no used generic type at the object model. The following example explains this problem. We consider a Web service operation `Op` which uses a generic type `Generic<String>` as an input parameter. The schema generation at the server side followed by a schema compilation at the client side will create a required service operation `Op` with `Generic<Object>` as an input parameter. Figure 10.2 illustrates the difference between the two interfaces at the client and at the server. The problem is due to a weakness in `XML` schema to represent the correspondent service generic type. In the following, we present some parts of the generated `WSDL` interface in order to clarify the problem:

```
...
<!-- wsdl input message declaration of the service operation -->
  <wsdl:message name="operation">
    <wsdl:part name="parameters" element="tns:operation">
    </wsdl:part>
  </wsdl:message>
  ...
  <!-- Description of the input parameter type, which corresponds
     to AGeneric<T> -->
  <xs:complexType name="operation">
    <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="tns:Generic"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Generic">
    <xs:sequence>
      <xs:element minOccurs="0" name="t" type="xs:anyType"/>
    </xs:sequence>
  </xs:complexType>
```

By a simple verification of the generated WSDL, the information about using String as an initialization of T type in Generic<T> type is completely missed. Thus, it is normal that the generated client code misses also this information.

By considering such a client interface, the client can associate any type for the "t" attribute. If the client calls the service operation with an Integer instance for example, an exception will be launched at server. Based on our tests on cxf 2.5.x, the moment of detecting this error and the type of the generated exception differs between SOAP and RESTful technologies:

- SOAP: the message will be unmarshalled at its receipt as an Integer instance. An exception will be launched only when we call the Integer instance as a String instance.
- RESTful: the received message will be associated to Null at unmarshalling. An exception is launched only if we call an operation on this gotten Null.

In order to make this distributed OO implementation near to its equivalent local implementation, the error must be detected from compilation. This corresponds to a correct generation of the client required interface conforming to the provided one by the server. Future work may define a full correct schema generation by resolving this problem and maybe other potential problems.

10.4 Improving RMI and CORBA with a service layer

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) [81] and Remote Method Invocation (RMI) [83]. RMI and CORBA lack a service layer: (i) objects are translated into a serialized form, with a binary format, instead of a document, with a structural format in RMI, (ii) and they are passed by references in CORBA.

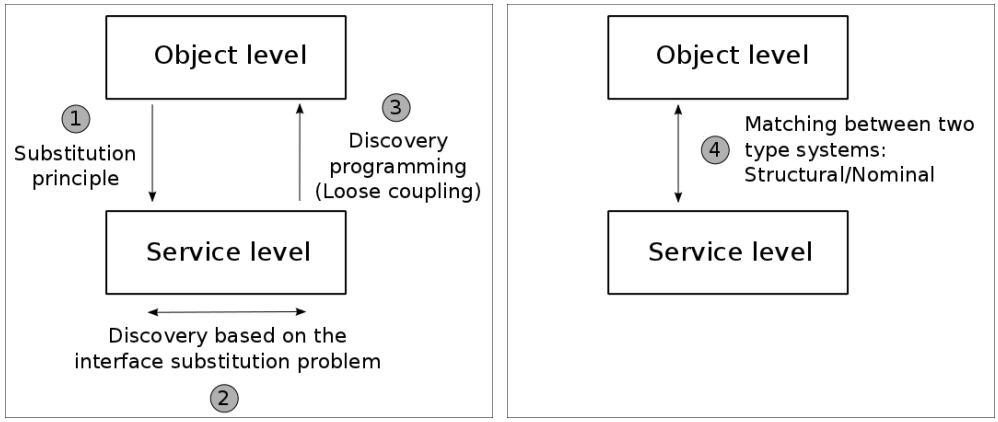
Since the failure of the substitution principle in object-oriented frameworks for Web services comes from the data binding required between the object layer and the service one, we can conclude that neither RMI nor CORBA provide a solution to this problem. Future work may apply our specification to RMI and CORBA in order to add a service layer with respect to the substitution principle.

10.5 Making Web services a real distributed object environment

Web services, as currently conceived, will not suffice to make a real distributed OO environment, since they mainly lack object references and lifecycle management [82]. One might argue that these uses are not what the Service-Oriented Architecture is intended to support. However, operators of Web-based direct sales systems are turning to use Web services to provide their applications and they want distributed objects. Therefore, it is time for the Web services community to start defining a foundation to improve the Web services architecture in order to reply the needs of their customers. RMI and CORBA technologies could be a good inspiration to take the Web Services architecture to that new level [14]. The challenge in such new architecture is to make such systems work reliably [15].

10.6 Implementing our specification in other frameworks

In this thesis, we have defined core and new requirements to be respected by all OO frameworks for Web services. We have focused only on the cxf framework, for making tests and implementing the specification. Future work can verify and implement this specification on other frameworks, like RestEasy and Systinet. It will be also interesting to study how the control flow works for other framework. The one presented in this thesis is deduced from our tests and studies on cxf. Moreover, our specification 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 this thesis 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. Future work may show how the problem can be resolved when different frameworks and OO programming languages are used.



(a) Problems discussed in this thesis (b) Future work

Figure 10.3: Matching the object level and the service level

10.7 Matching type systems between the object level and the service level

In this thesis, we discussed three problems in matching the object and the service levels which compose an OO framework for Web services (see Figure 10.3(a)):

1. Projecting the substitution principle from the object level to the service level in its two forms: value substitution and interface substitution,
2. Defining a structural type system with dynamic discovery and subtyping at the service level in order to allow the application of the interface substitution problem,
3. Projecting the discovery principle from the service level to the object level by unifying concepts in respect with loose coupling.

Another important issue exist between the two levels, it is related to the differences in the type systems (see Figure 10.3(b)):

- a nominal type system at the object level,
- a structural type system at the service level.

Malayeri et al. in [53] have discussed the integration of nominal and structural subtyping. In the inspiration of this work, it is useful in future work to study how to integrate the expressive structural type system presented in this thesis with the object type system, particularly for subtyping rules.

Appendix

Marshalling and Unmarshalling Phases in `cxf`

Classes in Figures A.1 and A.2 match to the following implementation classes in `cxf`:

- **SOAP:**
  - at unmarshalling:
    - the `UnmarshallingInteraceptor` matches with `DocLiteralnInterceptor`[1],
    - the `Reader` is an implementation class of `DataReader<T>`[2]. The default one is `DataReaderImpl<T>`[3],
    - the `DataBinder` is the plugged data binder to `cxf`. The default one is `JAXB`. Concretely, for `JAXB`, the `DataBinder` is the `JAXBContext` class[4]. For `JAXB`, the `Unmarshaller` class is an implementation of `Unmarshaller` interface[5]. `cxf` uses the `com.sun.xml.bind.v2.runtime.unmarshaller.UnmarshallerImpl` implementation class,

[1] https://cxf.apache.org/javadoc/latest/org/apache/cxf/interceptor/DocLiteralInInterceptor.html
[2] http://cxf.apache.org/javadoc/latest/org/apache/cxf/databinding/DataReader.html
[3] http://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxb/io/DataReaderImpl.html
[4] http://docs.oracle.com/javaee/5/api/javax/xml/bind/JAXBContext.html
[5] http://docs.oracle.com/javaee/5/api/javax/xml/bind/Unmarshaller.html

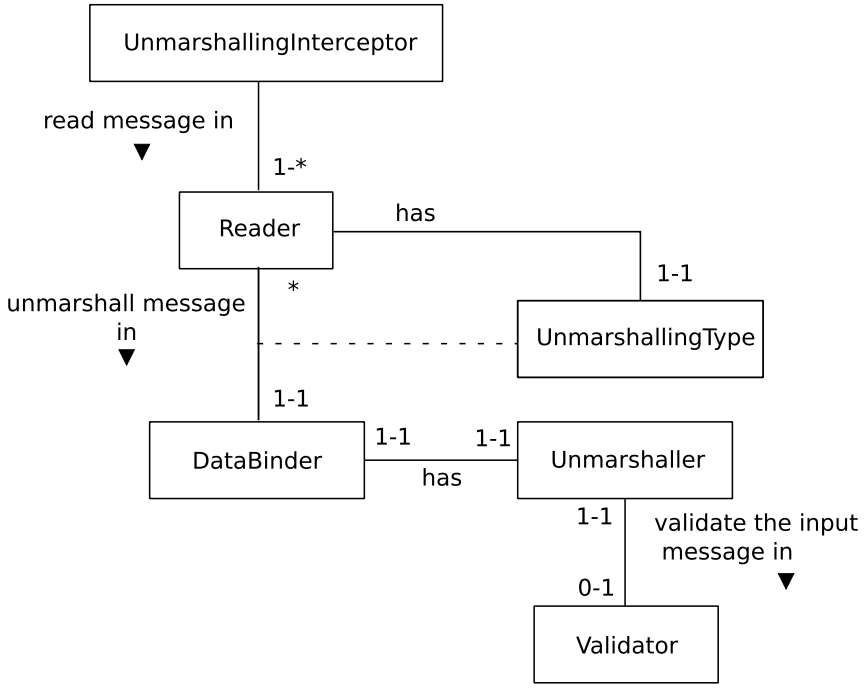


Figure A.1: Abstract UML of the unmarshalling phase in `cxf`

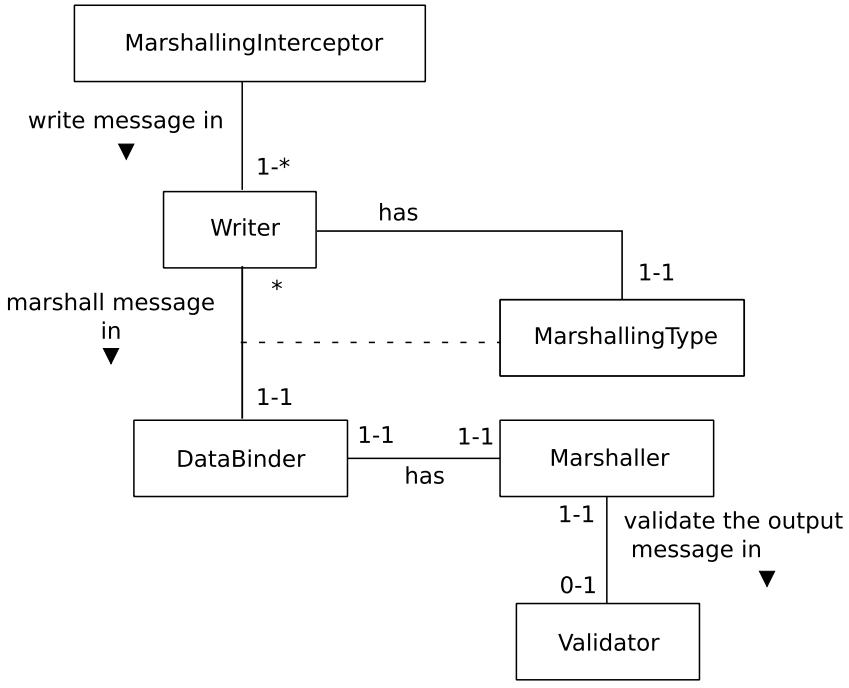


Figure A.2: Abstract UML of the marshalling phase in `cxf`

* the `Validator` depends on the plugged data binder. For `JAXB`, there are two forms:
  - schema validation: it is the validation conformally to a defined schema in the XSD. In the following we present an example of enabling the schema validation on the `Unmarshaller`:

```
// creation of a schema object referring to a "
   schema.xsd" file
SchemaFactory sf = SchemaFactory.newInstance(
   XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Schema schema = sf.newSchema(new
               File("schema.xsd"));

            // creation of the JAXBContext
               class for a unmarshalling
               following "MyClass" object type
            JAXBContext jc = JAXBContext.
               newInstance(MyClass.class);

            // creation of the unmarshaller
            Unmarshaller unmarshaller = jc.
               createUnmarshaller();

            // set the schema as a validator
               for the unmarshaller
             unmarshaller.setSchema(schema);
             MyClass c = (MyClass) unmarshaller
                 .unmarshal(new File("input.xml"
                 ));
```

  - `JAXB` validation: it is the default `JAXB` validation based on the schema binding (often represented as annotations on the object types) without the need to refer to a specific schema file. It is a subset of the schema validation that can be done with very little overhead. For the most part, that is just things like unknown elements or elements in wrong namespaces and such. The full schema validation can handle things like facets (minOccurs/maxOccurs, patterns, etc.). By default, the `Unmarshaller` does not validate. In order to activate the default `JAXB` validation, it is sufficient to call the `setEventHandler` operation with null argument on the `Unmarshaller`. It is also possible to set a new validation handler, an instance of class implementing `ValidationEventHandler` interface[6]. The ValidationEventHandler will be called by the `JAXB` Provider if any validation errors are encountered during calls to any of the unmarshal methods.

[6] http://docs.oracle.com/javaee/5/api/javax/xml/bind/ValidationEventHandler.html

In the following, we present an example of enabling the JAXB validation:

```
// creation of the JAXBContext class for a unmarshalling following "MyClass" object type
    JAXBContext jc = JAXBContext.newInstance(MyClass.class);

    // creation of the unmarshaller
    Unmarshaller unmarshaller = jc.createUnmarshaller();

    // set an instance of MyValidationEventHandler as validator to the unmarshaller
    unmarshaller.setEventHandler(new MyValidationEventHandler());
    MyClass c = (MyClass) unmarshaller.unmarshal(new File("input.xml"));
```

- at marshalling:
  - the `MarshallingInterceptor` matches with `BareOutInterceptor`[7],
  - the `Writer` is an implementation class of `DataWriter<T>`[8]. The default one is `DataWriterImpl<T>`[9],
  - the `DataBinder` is the plugged data binder to `cxf`. The default one is `JAXB`. For `JAXB`, the `Marshaller` class is an implementation of `Marshaller` interface[10],
  - the `Validator` is the same as defined before but applied on the `Marshaller`.

- **`RESTful`**
  - at unmarshalling:
    - the `UnmarshallingInteraceptor` matches with `JAXRSInInterceptor`[11],

[7] https://cxf.apache.org/javadoc/latest/org/apache/cxf/interceptor/BareOutInterceptor.html
[8] http://cxf.apache.org/javadoc/latest-2.6.x/org/apache/cxf/databinding/DataWriter.html
[9] http://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxb/io/DataWriterImpl.html
[10] http://docs.oracle.com/javaee/5/api/javax/xml/bind/Marshaller.html
[11] https://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/interceptor/JAXRSInInterceptor.html

    * the `Reader` is an implementation class of `javax.ws.rs.ext.MessageBodyReader<T>`[12]. The default one for `XML` is `JAXBElementProvider<T>`[13] and for `JSON` is `JSONProvider`[14].
    * the `DataBinder` is the same as for the `SOAP` case,
    * the `Validator` is the same as for the `SOAP` case.
  - at marshalling:
    * the `MarshallingInteraceptor` matches with `JAXRSOutInterceptor`[15],
    * the `Writer` is an implementation class of `javax.ws.rs.ext.MessageBodyWriter<T>`[16]. The default one for `XML` is `JAXBElementProvider<T>`[17] and for `JSON` is `JSONProvider`[18].
    * the `DataBinder` is the same as for the `SOAP` case,
    * the `Validator` is the same as for the `SOAP` case.

[12] http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/MessageBodyReader.html
[13] https://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/provider/JAXBElementProvider.html
[14] http://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/provider/json/JSONProvider.html
[15] https://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/interceptor/JAXRSOutInterceptor.html
[16] http://docs.oracle.com/javaee/6/api/javax/ws/rs/ext/MessageBodyReader.html
[17] https://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/provider/JAXBElementProvider.html
[18] http://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/provider/json/JSONProvider.html

B

Using Systinet Java API for UDDI Discovery

In the following listing, we present a code to implement the UddiRegistry class [1] using the Systinet Java API.

```
public class UDDIRegistry {
    UDDIProxy uddiProxy;
    public UDDIRegistry(String inquiryPort) {
       // Instantiate a local proxy of the UDDI server
       uddiProxy = new UDDIProxy(inquiryPort);
    }
    public <T> T  lookup (TModelKey tModelKey, Class<T> proxyInterface)
      throws UDDIException, LookupException {
        // Get WSDL from tModel
        String wsdlURI =
        uddiProxy.get_tModelOverviewURL(tModelKey.getValue()).getValue();
        // Get all services that have binding templates associated with
           tModel
        ServiceInfo[] serviceInfoArray =
         uddiProxy.find_service(null, null, null, tModelKey.getValue(),
             null);
        String serviceKey = serviceInfoArray[0].getServiceKey().getValue()
           ;
        // Get all binding templates associated with service and tModel
        BindingTemplate[] bindingTemplateArray =
          uddiProxy.find_binding(serviceKey, tModelKey.getValue(), null);
        ServiceClient serviceClient = ServiceClient.create(wsdlURI);
        // for each binding template, get the access point and try to bind
```

[1] http://www.drdobbs.com/web-development/uddi-dynamic-web-service-discovery/184405551

```
        for (int i=0; i < bindingTemplateArray.length; i++) {
22          String accessPoint =
                    bindingTemplateArray[i].getAccessPoint().getValue();
24          serviceClient.setServiceURL(accessPoint);

26          try {
               T ServiceProxy =
28                                  (T) serviceClient.createProxy(
                                       proxyInterface);
                System.out.println("Success");
30              return ServiceProxy;
            } catch (LookupException le) {
32              System.out.println("Unable to bind to
                                    service at " + accessPoint);
34          }
        }
36      throw new LookupException();
    }
38  }
```

C

Type Safety with Weak Authentication

Contents

In this appendix, we develop the authentication mechanism in two steps: we first consider systems that only use secure channels and then present the general case that includes the use of insecure channels. Finally, we review the type soundness property in presence of attackers.

C.1 Systems with secure channels only

As a first step we consider systems in which all channels are secure and do not modify messages. Since all channels are secure, nonces are not used in this step. The weak authentication protocol is defined in terms of the monitors (set $\mathcal{M}$) and the uncontrolled agents (set $\mathcal{U}$). The notation $\Theta^{\mathcal{U}}$ will denote the subset of a typing context containing only channels defined by an uncontrolled agent. The rules for the monitors are given in Figures C.1. Uncontrolled agents may modify messages: their values, channels and argument values may change arbitrarily. Furthermore, we assume that an uncontrolled agent knows all the provided channels in the aether, see [4] for the corresponding simple rules.

Monitors share a secret information $\mathtt{s}$, which is assumed to be *computation-resistant* [57]. At each instant a monitor can use $\mathtt{s}$ in its messages, but uncontrolled agents cannot use it and furthermore they cannot compute it from the received information. The set of tag values is defined as $TagValue = \{\mathtt{s}\} + \{\bot\}$. In Figure C.1, four rules define the behavior of a monitor

$$[\text{OUT}]_{\mathcal{M}\to\mathcal{M}} \qquad a[k^o(v)]\,,a[\Theta^o]\,,a_r[\Theta^\iota] \longrightarrow k(v,\_,\mathtt{s})^{true}\,,a[\Theta^o]\,,a_r[\Theta^\iota]$$
$$a,a_r \in \mathcal{M} \wedge k^\iota{:}{<}r{>} \in \Theta^\iota \wedge k^o{:}{<}t{>} \in \Theta^o \wedge [\![\, t;\, v \,]\!] \wedge \Theta^o \mid_{\mathrm{K}(v)} \leq\ \Pi(t,v)$$

$$[\text{OUT}]_{\mathcal{M}\to\mathcal{U}} \qquad a[k^o(v)]\,,a[\Theta^o]\,,a_r[\Theta^\iota] \longrightarrow k(v,\_,\bot)^{true}\,,a[\Theta^o]\,,a_r[\Theta^\iota]$$
$$a \in \mathcal{M} \wedge a_r \in \mathcal{U} \wedge k^\iota{:}\ r \in \Theta^\iota \wedge k^o{:}{<}t{>} \in \Theta^o \wedge [\![\, t;\, v \,]\!] \wedge \Theta^o \mid_{\mathrm{K}(v)} \leq\ \Pi(t,v)$$

$$[\text{IN}]_{\mathcal{M}\to\mathcal{M}} \qquad k(v,\_,\mathtt{s})^{true}\,,a[\Theta^\iota]\,,a[\Theta^o] \longrightarrow a[k^\iota(v,true)^{true}]\,,a[\Theta^\iota]\,,a[\Theta^o\ \bar{\wedge}\ \Pi(t,v)]$$
$$a \in \mathcal{M} \wedge k^\iota{:}{<}t{>} \in \Theta^\iota \wedge [\![\, t;\, v \,]\!]$$

$$[\text{IN}]_{\mathcal{U}\to\mathcal{M}} \qquad k(v,\_,\bot)^{false}\,,a[\Theta^\iota]\,,a[\Theta^o] \longrightarrow a[k^\iota(v,false)^{false}]\,,a[\Theta^\iota]\,,a[\Theta^o\ \bar{\wedge}\ \Pi(t,v)^{\mathcal{U}}]$$
$$a \in \mathcal{M} \wedge k^\iota{:}{<}t{>} \in \Theta^\iota \wedge [\![\, t;\, v \,]\!]$$

Figure C.1: Secure channels, monitors

when sending messages to, or receiving messages from, other monitors or uncontrolled agents ($\mathcal{M}$ and $\mathcal{U}$ in the rule names). Compared to be basic scheme shown in Figure 6.4, one modification is that a message from a monitor to a monitor incorporates the secret, and at receipt by a monitor the presence of the secret is checked. Message sent to an uncontrolled agent do not need a secret.

To prove the correctness of this authentication scheme we have to compare the status (true or false) inferred from the checking of the secret (*i.e.,* the sticky value component in the messages) with the true status of the emitter that we note using superscript boolean values. All messages, emitted and received ones, are annotated either with true ($true$) denoting a controlled emitter or false ($false$) that characterizes an uncontrolled one. Hence, the rules take into account the status computation as follows: *i)* $[\text{OUT}]_{\mathcal{M}\to *}$ the status of the emitted message is $true$ since the emitter is a monitor, *ii)* $[\text{OUT}]_{\mathcal{U}\to *}$ the status of the emitted message is $false$, and *iii)* $[\text{IN}]_{*\to *}$ keep the status of the received message. Informally, correctness of the authentication scheme is defined by the following property: for each message received from a monitor, the real status is the same as the computed one.

Theorem 2 (Weak Message Authentication)

$$\forall\, a \in \mathcal{M}\,.\,\forall\, k,\ v,\ b,\ r,\ st\,.\,a[k^\iota(v,b)^{st}] \in \Omega \implies b = st$$

Proof. Considering the incoming message $a[k^\iota(v,b)^{st}]$, where $a \in \mathcal{M}$, two subcases have to be handled depending on the value of b. If $b = true$ then the message was received according to the rule $[\text{IN}]_{\mathcal{M}\to\mathcal{M}}$ and the transmitted message was $k(v,-,\mathtt{s})^{st}$ with $\neg(\mathtt{s} = \bot)$. The only applicable rule then is $[\text{OUT}]_{\mathcal{M}\to\mathcal{M}}$, the emitter was a monitor and $st = true$. If $b = false$ then the rule $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ was used, and the rule $[\text{OUT}]_{\mathcal{U}\to\mathcal{M}}$ emitted the message, thus $st = false$. □ ■

C.2 Systems with insecure channels

We now consider the generalization of the authentication scheme to systems with insecure channels. An insecure channel between two monitors is equivalent to the transmission of messages via uncontrolled agents which act as malicious routers. The challenge is to force the uncontrolled agents to act with integrity. When a message from a monitor reaches an uncontrolled agent, the latter may suppress it or send another message to monitors or uncontrolled agents. We use hash keys in messages to be able to detect modifications on messages. The behavior of monitors has to be extended so that messages going to or coming from uncontrolled agents are checked for integrity violations. We also need to review the status computation.

The new rules should express the specific behavior of routers, which are not controlled but: *i)* receive messages with correct hash keys from monitors, and *ii)* if the hash key is copied with the right values, the status component of the outgoing message is the $true$ status, or else it is $false$. We use nonces to avoid that a same message will be received several times. Otherwise, agents could replay messages. Nonces are generated from an injective function of the agent. This implies that emitted messages now have a mandatory integer argument for nonces. Furthermore, the input and output interfaces of an agent store the known channels with their nonces.

We assume that monitors share a secret cryptographic function (H) making the secrets unforgeable by uncontrolled agents. We need to distinguish the hashed values from the secret s and $\perp$. The set of tag values is thus redefined to $TagValue = \{\perp\} + \{\mathtt{s}\} + range(\mathtt{H})$.

Figure C.2 adds two new rules related to insecure message transit and change the rules for uncontrolled agents to take into account the modified status computation. In this figure, $*$ denotes any term. We have new materials to manage nonces, channels and interfaces are enriched to cope with that. The notation $\Theta^o(k, +1)$ stands for the incrementation of the nonce counter for channel k in the Θ^o interface. $\Theta^\iota(k, n)$ adds the new n in the set of nonces N already seen by the channel k in the input interface Θ^ι $(k^\iota{:}{<}t{>}, (N))$.

Rule $[\text{OUT}]_{\mathcal{M}\to\mathcal{U}}$ allows a monitor a to send a message to another monitor m via an uncontrolled agent r owning the channel k_r. To do this a hash value $H(v, k, n)$ is added to the message. Rule $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ formalizes the case where the hash key value received by a monitor is compliant with the channel k, the value v and the nonce n. Rule $[\text{OUT}]_{\mathcal{U}\to *}$ specifies that the original emitter was a monitor if the hash value is correct. The remaining rules are immediate.

The correctness of the extended authentication scheme is formulated as a theorem analogous to the first scheme for secure channels. Its proof has to take into account that messages may be forwarded by uncontrolled routers, see Figures C.1 and C.2. Hence, it is possible that messages with hash keys are routed in complex manners within the uncontrolled world.

Proposition 3 (Routing Control) *Any finite message sequence in the aether $k_i(v_i, n_i, \mathtt{H}_i)$ that is temporally ordered, such that $\exists\, H\, .\, \forall\, i\, .\, \mathtt{H}_i = H$, starts from a monitor.*

Proof. Using the rule Rule $[\text{OUT}]_{\mathcal{M}\to\mathcal{U}}$ a monitor can produce such a message. Hash key values are not forgeable by uncontrolled agents, uncontrolled agents can only pass them to others. A monitor could not forge a message with a copy of an hash key value, since we are using nonces associated to channels and agents and H is injective. Thus any hashed message has to be initially

$$[\text{OUT}]_{\mathcal{M}\to\mathcal{U}} \quad \frac{a[k^o(v)],\ a[\Theta^o]\,,\ a_r[\Theta^\iota]}{k(v,n,\mathtt{H}(v,k,n))^{true},\ a[\Theta^o(k,+1)]\,,\ a_r[\Theta^\iota]} \quad a,m\in\mathcal{M}\wedge a_r\in\mathcal{U}\wedge k^\iota{:}r\in\Theta^\iota\wedge k^o{:}<t>\in\Theta^o\wedge[\![\,t;\,v\,]\!]\wedge\Theta^o\mid_{\mathrm{K}(v)}\leq\Pi(t,v)$$
$$k^o{:}<t_m>,(n)\in\Theta^o\wedge[\![\,t_m;\,v\,]\!]\wedge\Theta^o\mid_{\mathrm{K}(v)}\leq\Pi(t_m,v)$$

$$[\text{IN}]_{\mathcal{U}\to\mathcal{M}} \quad \frac{k(v,n,\mathtt{H}(v,n,k))^{true}\,,\ a[\Theta^\iota]\,,\ a[\Theta^o]}{a[k^\iota(v,true))^{true}]\ a[\Theta^\iota(k,n)]\,,\ a[\Theta^o\ \bar{\wedge}\ \Pi(t,v)]} \quad a\in\mathcal{M}\wedge k^\iota{:}<t>,(N)\in\Theta^\iota\wedge n\notin N\wedge[\![\,t;\,v\,]\!]$$

$$[\text{IN}]_{\mathcal{U}\to\mathcal{M}} \quad \frac{k(v,n,h)^{false}\,,a[\Theta^o]}{a[k^\iota(v,false)^{false}]\,,a[\Theta^o\ \bar{\wedge}\ \Pi(t,v)^{\mathcal{U}}]} \quad a\in\mathcal{M}\wedge\neg(h=\mathtt{H}(v,n,k))\wedge[\![\,t;\,v\,]\!]$$

$$[\text{OUT}]_{\mathcal{U}\to *} \quad \frac{a[*^o(*)^{true|false}]}{k(*)^{h=\mathtt{H}(v,n,k)}} \quad a\in\mathcal{U}$$

$$[\text{IN}]_{*\to\mathcal{U}} \quad \frac{k(*)^{true|false}}{a[*^\iota(*)^{true|false}]} \quad a\in\mathcal{U}$$

Figure C.2: Weak Message Authentication Rules – Insecure Channels

emitted from a monitor and intermediate uncontrolled nodes cannot forge but always relay the hash value. □ ■

A consequence is that all the messages in the sequence have status $true$. Furthermore, messages with hash key are received at most once due to the nonce mechanism. Overall, this scheme amounts to the use of so-called time-variant parameters [57]. While weak message authentication (Theorem 2) still holds in the presence of uncontrolled agents, its proof has to be adapted.

Proof. (of Theorem 2 in the presence of uncontrolled agents). Consider a message that enters a monitor $a[k^\iota(v,b)^{st}]$, where $a\in\mathcal{M}$, $k\in\Theta^\iota$.

- If $b=true$ the accepting rule was $[\text{IN}]_{\mathcal{M}\to\mathcal{M}}$ and we are in the same situation as with only secure channels. The accepting rule might also have been $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$. In this case the monitor receives a correct hashed key message $k(v,n,\mathtt{H}(v,n,k))$. Property 3 states that the original hash value was emitted by a monitor and the status is $true$.

- If $b=false$ either the rule was $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ but this rule exists in two instances: one with $\perp$ and the other with a hash key h. The first case is the same as previous with secure channel. In the second case h is not a correct hash key value, thus it was putted in the aether by $[\text{OUT}]_{\mathcal{U}\to *}$ and then the status is $false$. □

■

C.3 Type soundness with attackers

A countermeasure to possible attacks on types is to add control at reception time during channel discovery.

Definition 7 (Origin-check) *At message reception time, the type inference mechanism checks the origin of each message (using weak message authentication). If the message was sent by an uncontrolled agent, no discovery of channels created by monitors is allowed.*

We must take care of the discovery of the channels defined by monitors. Channels defined by uncontrolled agents are not trusted. This countermeasure is formalized in the rules $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ in Figures C.1 (and C.2 in the appendix). When the computed status is $false$, no channel defined by a monitor is discovered.

Theorem 3 (Invariants) *In presence of uncontrolled agents and insecure channels, well-formedness and interface consistency are preserved.*

The proof is immediate from the origin-check. Since we have a context with secure and insecure agents and channels we need to review the formulation and proof of the soundness property.

Theorem 4 (Soundness with attackers) *If processes are well-formed and interfaces consistent:*

$$\forall\, k, v, n\,.\,(k(v,-,s) \in \Omega \vee k(v,n,\mathtt{H}(v,k,n)) \in \Omega) \implies k(v) \textit{ is correct}$$

Proof. Since we use the origin-ckeck countermeasure, well-formedness and interface consistency are invariants. We have two distinct kinds of messages depending if the sticky value sv is the secret `s` or a hash value. In the first case we have a similar proof than for theorem 1. In the the second case, the routing control Proposition 3 states that there was an initial message $k_r(v, n, \mathtt{H}(v, k, n))$ from a monitor. The emitting rule checks the type of v relatively to the final m monitor target and as in the fully secure case we have interface consistency and monotony thus inference succeeds at the receipt site. ■

D

Details About the SOAP / RESTful Interfaces and Message Exchanges for the Flight Reservation Scenario

Contents

D.1 Detailed WSDL file of the flight reservation scenario

```
<definitions>
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  TYPE DEFINITION - List of types
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~-->
  <types>
    <schema .../>
      <element name="request">
        <complexType>
          <sequence>
            <element name="source" type="string"/>
```

```
            <element name="destination" type="string"/>
            <element name="date" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="reply">
        <complexType>
          <sequence>
            <element name="id" type="string" minOccurs = "0" maxOccurs = "
                unbounded"/>
          </sequence>
        </complexType>
      </element>
      <element name="travelId" type= "string"/>
      <element name="confirmation" type="boolean"/>
    </schema>
  </types>

  <!--
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  MESSAGE TYPE DEFINITION - Definition of the message types
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <message name="searchRequest">
    <part name="travelRequest" element="tns:request"/>
  </message>
  <message name="searchReply">
    <part name="travelReply" element="tns:reply"/>
  </message>
  <message name="bookingRequest">
    <part name="bookingRequest" element="tns:travelId"/>
  </message>
  <message name="bookingReply">
    <part name="bookingConfirmation" element="tns:confirmation"/>
  </message>

  <!--
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  PORT TYPE DEFINITION - A port type groups a set of operations into
  a logical service unit.
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <portType name="FlightTravelServicePT">
    <operation name="searchTravel">
      <input message="searchRequest" />
      <output message="searchReply" />
    </operation>
    <operation name="bookTravel">
      <input message="bookingRequest" />
      <output message="bookingReply" />
    </operation>
  </portType>
```

```
  <binding name="flightTravelServiceBinding">
    <!-- details of binding description are omitted here-->
    <portType name = "FlightTravelServicePT" />
  </binding>

  <service name="flightTravelService">
    <port name="flightTravelPort" binding="travelServiceSOAPBinding">
      <address location="http://flight-travel-service"/>
    </port>
  </service>
</definitions>
```

D.2 Detailed WADL file of the flight reservation scenario

```
<application>
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  TYPE DEFINITION - List of types
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~->
    <grammars>
      <element name="reply">
    <complexType>
      <sequence>
        <element name="id" type="string" minOccurs = "0" maxOccurs = "
            unbounded"/>
      /sequence>
    </complexType>
  </element>
  <element name="confirmation" type="boolean"/>
    </grammars>
      <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Root resource
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~->
    <resources base="[http://flight-travel-service/">
        <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Travel sub-resource
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~->
      <resource path="travel/">
          <method name="GET">
              <request>
                  <param type="string" style="query" name="source"/>
                  <param type="string" style="query" name="destination"/
                      >
                  <param type="string" style="query" name="date"/>
              </request>
              <response>
                  <representation mediaType="application/xml" element= "
                      reply" />
              </response>
```

```
            </method>
        </resource>
        <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Booking sub-resource
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~->
        <resource path="booking/">
            <method name="Put">
                <request>
                    <param type="string" style="query" name="travelId"/>
                </request>
                <response>
                    <representation mediaType="application/xml" element="
                        confirmation"/>
                </response>
            </method>
        </resource>
    </resources>
</application>
```

D.3 Reduction rules for booking a flight travel

In the following, we present the succession of chemical rules to book a flight travel. For simplification reasons, we represent the `bookingRequest` type by t_1 and the `bookingConfirmation` type by t_2 defined as:

$$t_1 = \texttt{bookingRequest} = \texttt{travel[id[string], End], End}$$
$$t_2 = \texttt{bookingConfirmation} = \texttt{confirmation[bool], End}$$

$$
\begin{array}{lll}
[\mathrm{LOC}]_{\gamma_{client}} & \gamma_{client}[\sigma_{Client_0}] & \longrightarrow \quad \gamma_{client}[k_{bookTravel}(travel(id(\text{"}flight02\text{"})), k_{bookReply})], \\
 & & \quad \gamma_{client}[\sigma_{Client_1}] \\
[\mathrm{OUT}]_{\gamma_{client}\rightarrow\gamma_{server}} & \gamma_{client}[k_{bookTravel}(travel(id(\text{"}flight02\text{"})), k_{bookReply})], & \longrightarrow \quad k_{bookTravel}(travel(id(\text{"}flight02\text{"})), k_{bookReply}), \\
 & \gamma_{client}[k^{o}_{bookTravel} : t_1], & \quad \gamma_{client}[k^{o}_{bookTravel} : t_1], \\
 & \gamma_{client}[k^{\iota o}_{bookReply} : t_2] & \quad \gamma_{client}[k^{\iota o}_{bookReply} : t_2] \\
[\mathrm{IN}]_{\gamma_{client}\rightarrow\gamma_{server}} & k_{bookTravel}(travel(id(\text{"}flight02\text{"})), k_{bookReply}), & \longrightarrow \quad \gamma_{server}[k_{bookTravel}(travel(id(\text{"}flight02\text{"})), k_{bookReply})], \\
 & \gamma_{server}[k^{\iota}_{bookTravel} : t_1] & \quad \gamma_{server}[k^{\iota}_{bookTravel} : t_1], \\
 & & \quad \gamma_{server}[k^{o}_{bookReply} : t_2] \\
[\mathrm{LOC}]_{\gamma_{server}} & \gamma_{server}[\sigma_{Server_0}], & \longrightarrow \quad \gamma_{server}[k_{bookReply}(confirmation(true))], \\
 & \gamma_{server}[k_{bookTravel}(travel(id(\text{"}flight02\text{"})), k_{bookReply})] & \quad \gamma_{server}[\sigma_{Server_1}] \\
[\mathrm{OUT}]_{\gamma_{server}\rightarrow\gamma_{client}} & \gamma_{server}[k_{bookReply}(confirmation(true))], & \longrightarrow \quad k_{bookReply}(confirmation(true)), \\
 & \gamma_{server}[k^{o}_{bookReply} : t_2] & \quad \gamma_{server}[k^{o}_{bookReply} : t_2] \\
[\mathrm{IN}]_{\gamma_{server}\rightarrow\gamma_{client}} & k_{bookReply}(confirmation(true)), & \longrightarrow \quad \gamma_{client}[k_{bookReply}(confirmation(true))], \\
 & \gamma_{client}[k^{\iota o}_{bookReply} : t_2] & \quad \gamma_{client}[k^{\iota o}_{bookReply} : t_2] \\
[\mathrm{LOC}]_{\gamma_{server}} & \gamma_{client}[\sigma_{Client_1}], & \\
 & \gamma_{client}[k_{bookReply}(confirmation(true))] & \longrightarrow \quad \gamma_{client}[\sigma_{Client_2}]
\end{array}
$$

Figure D.1: Succession of Client/Server scenario reduction rules

Details about our Adaptations to `cxf`

Contents

E.1 `cxf` configuration to apply the lifting algorithm

In this section, we present how to configure `cxf` to resolve the existing interoperability problems due to the application of the substitution principle, for both `SOAP` and `RESTful` models.

E.1.1 Configuration for `SOAP`

In order to disable the structural type validation in its two forms, `JAXB` validation and schema validation (defined in A for the `validator` component), the `Reader` has to call the `setSchema` with a **null** argument on the unmarshaller. In order to achieve this end, `cxf` offers a way to

configure JAX-WS properties, which are used by the Reader, at the client and the server using a spring file[1]. The spring file at the server and the client sides should define the following properties:

```
<jaxws:properties>
  <entry key="schema-validation-enabled" value="false" />
  <entry key="set-jaxb-validation-event-handler" value="false" />
</jaxws:properties>
```

The spring configuration file should be referenced in the servlet configuration file at the server. While, the client should be created using the spring configuration file, eg. "beans.xml", as follows:

```
ClassPathXmlApplicationContext context
      = new ClassPathXmlApplicationContext(new String[] {"client/
         beans.xml"});

      // getting a client proxy for the required service
         interface
      ServiceInterface client = (ServiceInterface)context.getBean
         ("client");
```

E.1.2 Configuration for RESTful

By default, cxf does not enable schema validation for RESTful. In order to call an unmarshal operation with a declared type, the Reader has to use the unmarshalling object type as a second parameter of the unmarshal operation. For this end, it is sufficient to set the unmarshalAsJaxbElement filled to "true" in the JAXBElementProvider instance (the default JAX-RS Reader for XML) and in the JSONProvider instance (the default JAX-RS Reader for JSON). These two classes extend the abstract class AbstractJAXBProvider[2] which defines the unmarshalAsJaxbElement field. For that, cxf offers a way to configure JAX-RS at the client and the server using a spring file[3]. The spring file at the client and the server sides should define the beans for the JAXBElementProvider and the JSONProvider as follows:

```
<bean id="xmlProvider" class="org.apache.cxf.jaxrs.provider.
   JAXBElementProvider">
  <property name="unmarshallAsJaxbElement" value="true" />
</bean>
```

[1] http://cxf.apache.org/docs/jax-ws-configuration.html

[2] http://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/provider/AbstractJAXBProvider.html

[3] http://cxf.apache.org/docs/jax-rs-data-bindings.html, https://cxf.apache.org/docs/jaxrs-services-configuration.html#JAXRSServicesConfiguration-beans.xml

```
<bean id="jsonProvider" class="org.apache.cxf.jaxrs.provider.json.
    JSONProvider">
  <property name="unmarshallAsJaxbElement" value="true" />
</bean>
```

As for SOAP, the spring configuration file should be referenced in the servlet configuration file at the server. While for the client, it should be created using the spring configuration file, eg. "beans.xml", as follows:

```
ClassPathXmlApplicationContext context
      = new ClassPathXmlApplicationContext(new String[] {"client/
          beans.xml"});

      // getting a client proxy for the required service
          interface
      ServiceInterface client = (ServiceInterface)context.getBean
          ("client");
```

E.2 cxf configuration to avoid root element mapping in RESTful

The JAX-RS implementation in cxf defines JSONProvider<T> and JAXBElementProvider<T> which are the reader and the writer for JSON and XML respectively. These two classes extend AbstractJAXBProvider<T>[4] which defines two boolean attributes marshalAsJaxbElement and unmarshalAsJaxbElement in order enable or disable the marshalling/unmarshalling as a JaxbElement. Therefore, using the cxf Spring configuration, the JAX-RS providers for XML and JSON should be configured as follows:

```
<bean id="xmlProvider" class="org.apache.cxf.jaxrs.provider.
    JAXBElementProvider">
  <property name="unmarshallAsJaxbElement" value="true" />
  <property name="marshallAsJaxbElement" value="true" />
</bean>
<bean id="jsonProvider" class="org.apache.cxf.jaxrs.provider.json
    .JSONProvider">
  <property name="unmarshallAsJaxbElement" value="true" />
  <property name="marshallAsJaxbElement" value="true" />
</bean>
```

[4] http://cxf.apache.org/javadoc/latest-2.7.x/org/apache/cxf/jaxrs/provider/AbstractJAXBProvider.html

E.3 cxf configuration to create a global JAXB context

In this section, we show how to configure cxf, in order to create a single and global JAXB instance for RESTful and SOAP services:

- for SOAP: We have to create a new class GlobalContext which creates a JAXBContext from all the service data classes. In the following, we show an example of this class for the previous example using op:

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import server.*;
public class GlobalContext {

  public static JAXBContext getJAXBContext(){
    try {
      return JAXBContext.newInstance(A.class, B.class, Op.class
          , OpResponse.class);
    } catch (JAXBException e) {
      e.printStackTrace();
    }
    return null;
  }

}
```

  Then, we should configure the "beans.xml" by adding a bean which calls the getJAXBContext operation of the GlobalContext class as follows:

```
<bean id="globalContext" class="server.serviceClasses.
   GlobalContext"
    factory-method="getJAXBContext" />
```

  Then we have to add a configuration to the JAX-WS (jaxws:endpoint at the server or jaxws:client at the client) data binding by specifying that for JAXB instantiation, the constructor of org.apache.cxf.jaxb.JAXBDataBinding should be referenced to the previous defined bean:

```
<jaxws:dataBinding>
  <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
    <constructor-arg index="0" ref="globalContext" />
  </bean>
</jaxws:dataBinding>
```

- for RESTful: The XML and JSON provider, as we have explained before for RESTful, should be configured as in the following for the example of op service operation:

```
<bean id="xmlProvider" class="org.apache.cxf.jaxrs.provider.
    JAXBElementProvider">
  <property name="extraClass">
    <!-- here is the list of all data classes -->
    <list>
      <value>server.A</value>
      <value>server.B</value>
    </list>
  </property>
  <property name="singleJaxbContext" value="true" />
  <property name="useSingleContextForPackages" value="true"
     />
</bean>
<bean id="jsonProvider" class="org.apache.cxf.jaxrs.provider.
    json.JSONProvider">
  <property name="extraClass">
    <!-- here is the list of all data classes -->
    <list>
      <value>server.A</value>
      <value>server.B</value>
    </list>
  </property>
  <property name="singleJaxbContext" value="true" />
  <property name="useSingleContextForPackages" value="true"
     />
</bean>
```

E.4 Merging all required adaptations

In the following, we present first a standard configuration of `cxf` to resolve all problems discussed in this thesis. Then we propose an automation algorithm of this configuration at the development of the service and the client.

E.4.1 `cxf` configuration

In the following we show how the `"beans.xml"` configuration file for a Web service should be configured for `SOAP` and `RESTful`, at the client and the server sides in order to merge our proposed solution for the substitution and loose coupling problems. We consider as an example, a service operation: $op : A \rightarrow void$, and a subclass `B` of class `A`.

- for `SOAP`:
  - `beans.xml` at the client side:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:jaxws="http://cxf.apache.org/jaxws" xmlns:xsi="http
     ://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/
             spring-beans.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd">
  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-
     soap.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-
     http.xml" />


  <jaxws:client id="client" serviceClass="client.
     serviceClasses.ServiceInterface"
    address="http://localhost:8080/SOAP-
       InterfaceSubstitutionWithInterfaces/services/
       ServicePort">
    <jaxws:properties>
      <entry key="schema-validation-enabled" value="false"
         />
      <entry key="set-jaxb-validation-event-handler" value=
         "false" />
    </jaxws:properties>
    <jaxws:dataBinding>
      <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
        <constructor-arg index="0" ref="globalContext" />
      </bean>
    </jaxws:dataBinding>
  </jaxws:client>
  <bean id="globalContext" class="client.serviceClasses.
     GlobalContext"
    factory-method="getJAXBContext" />
</beans>
```

Listing E.1: Configuration at the client side for SOAP

– `beans.xml` at the server side:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation="http://www.springframework.org/schema
    /beans http://www.springframework.org/schema/beans/
    spring-beans-2.5.xsd http://cxf.apache.org/jaxws http
    ://cxf.apache.org/schemas/jaxws.xsd">
  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/cxf/cxf-extension-
    soap.xml" />
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"
     />
  <jaxws:endpoint xmlns:tns="http://server/" id="service"
    implementor="server.serviceClasses.Service"
       wsdlLocation="wsdl/service.wsdl"
    endpointName="tns:ServicePort" serviceName="tns:
       ServiceService"
    address="/ServicePort">
    <jaxws:properties>
      <entry key="schema-validation-enabled" value="false"
         />
      <entry key="set-jaxb-validation-event-handler" value=
         "false" />
    </jaxws:properties>
    <jaxws:features>
      <bean class="org.apache.cxf.feature.LoggingFeature"
         />
    </jaxws:features>

    <jaxws:dataBinding>
      <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
        <constructor-arg index="0" ref="globalContext" />
      </bean>
    </jaxws:dataBinding>
  </jaxws:endpoint>
  <bean id="globalContext" class="server.serviceClasses.
    GlobalContext"
    factory-method="getJAXBContext" />
</beans>
```

Listing E.2: Configuration at the server side for SOAP

- for RESTful:
  - bean.xml at the client side:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xmlns:jaxws="http://cxf.apache.org/jaxws" xmlns:cxf="http
     ://cxf.apache.org/core"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.
   xsd
http://cxf.apache.org/jaxrs
http://cxf.apache.org/schemas/jaxrs.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd
http://cxf.apache.org/core
      http://cxf.apache.org/schemas/core.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml" />


  <jaxrs:client id="client"
    address="http://localhost:8080/REST-
       InterfaceSubstitutionWithInterfaces"
    serviceClass="client.serviceClasses.ServiceInterface">
    <jaxrs:providers>
      <ref bean="xmlProvider" />
      <ref bean="jsonProvider" />
    </jaxrs:providers>
  </jaxrs:client>
  <bean id="xmlProvider" class="org.apache.cxf.jaxrs.
     provider.JAXBElementProvider">
    <property name="extraClass">
      <!-- here is the list of all classes in the client
         package -->
      <list>
        <value>client.A</value>
        <value>client.B</value>
      </list>
    </property>
    <property name="singleJaxbContext" value="true" />
    <property name="useSingleContextForPackages" value="
       true" />
    <property name="unmarshallAsJaxbElement" value="true"
       />
    <property name="marshallAsJaxbElement" value="true" />
  </bean>
  <bean id="jsonProvider" class="org.apache.cxf.jaxrs.
     provider.json.JSONProvider">
```

```
    <property name="extraClass">
      <!-- here is the list of all classes in the client
         package -->
      <list>
        <value>client.A</value>
        <value>client.B</value>
      </list>
    </property>
    <property name="singleJaxbContext" value="true" />
    <property name="useSingleContextForPackages" value="
       true" />
    <property name="unmarshallAsJaxbElement" value="true"
       />
    <property name="marshallAsJaxbElement" value="true" />
  </bean>
</beans>
```

Listing E.3: Configuration at the client side for `RESTful`

– `beans.xml` at the server side:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xmlns:jaxws="http://cxf.apache.org/jaxws" xmlns:cxf="http
     ://cxf.apache.org/core"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.
   xsd
http://cxf.apache.org/jaxrs
http://cxf.apache.org/schemas/jaxrs.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd
http://cxf.apache.org/core
      http://cxf.apache.org/schemas/core.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml" />
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"
      />

  <jaxrs:server id="testRestful" address="/">
    <jaxrs:serviceBeans>
      <ref bean="myService" />
    </jaxrs:serviceBeans>
    <jaxrs:features>
```

```
      <cxf:logging />
    </jaxrs:features>
    <jaxrs:providers>
      <ref bean="xmlProvider" />
      <ref bean="jsonProvider" />
    </jaxrs:providers>
  </jaxrs:server>
  <bean id="myService" class="server.serviceClasses.Service
     " />
  <bean id="xmlProvider" class="org.apache.cxf.jaxrs.
     provider.JAXBElementProvider">
    <property name="extraClass">
      <!-- Here is the list of all classes in the server
         package -->
      <list>
        <value>server.A</value>
        <value>server.B</value>
      </list>
    </property>
    <property name="singleJaxbContext" value="true" />
    <property name="useSingleContextForPackages" value="
       true" />
    <property name="unmarshallAsJaxbElement" value="true"
       />
    <property name="marshallAsJaxbElement" value="true" />
  </bean>
  <bean id="jsonProvider" class="org.apache.cxf.jaxrs.
     provider.json.JSONProvider">
    <property name="extraClass">
      <!-- Here is the list of all classes in the server
         package -->
      <list>
        <value>server.A</value>
        <value>server.B</value>
      </list>
    </property>
    <property name="singleJaxbContext" value="true" />
    <property name="useSingleContextForPackages" value="
       true" />
    <property name="unmarshallAsJaxbElement" value="true"
       />
    <property name="marshallAsJaxbElement" value="true" />
  </bean>
</beans>
```

Listing E.4: Configuration at the server side for `RESTful`

E.4.2 An automation algorithm

By analyzing the previous presented "beans.xml" files, we can deduce the following standard configuration requirements that differs depending on the service or the client implementation:

- for SOAP:
  - at the client, there is a need to:
    - define an id for the client proxy, the required service interface, and the Web service url. These are the parameters of the jaxws:client XML element as presented in Listing E.1.
    - build a GlobalContext class defining a globalContext method which returns a JaxbContext using all the defined data classes in the developed code. That corresponds to the jaxws:dataBinding configuration presented in Listing E.1.
  - at the server, there is a need to:
    - define an id for the server endpoint, an implementation class, a WSDL location, an endpointName, a service name and a url address. These are the parameters of the jaxws:endpoint XML element as presented in Listing E.2.
    - build a GlobalContext class as described previously at the client side.
- for RESTful:
  - at the client, there is a need to:
    - define an id for the client proxy, the required service interface and the service url. These are the parameters of the jaxrs:client XML element as presented in Listing E.3.
    - define all the data classes in the developed code as a list of extra classes in the XML and JSON providers, see extraClass property defined in Listing E.3.
  - at the server, there is a need to:
    - define an id of the RESTful service, the service implementation class and a base url address. These are the parameters of the jaxrs:server XML element as presented in Listing E.4.
    - define all the data classes in the developed code as a list of extra classes in the XML and JSON providers as described previously at the client side.

In order to automate these standard configurations, we propose the following facilities at the server and the client sides, (we use Java as language for our abstract codes):

- for SOAP: At the server side, at the deployment of the Java code as a SOAP service, the following abstract code should be applied:

```
File deploy(String id, Class service, String wsdl, String
   endpointName, String serviceName, String URL){
  /**
   * Creating an adapter when an interface is used as parameter
       type or return type in a service method
   */
  Method[] mList = anInterface.getMethods();
  for (Method m:mList){
    Class<?>[] paramTypes = m.getParameterTypes();
    for(Class<?> c : paramTypes){
      if(c.isInterface()){
        Class adaptedInterface = GenerateAdaptedInterface(c);
        Class adapter = CreateAdapter(c, adaptedInterface);
        CreateAdapterAnnotation(c, adapter);
      }
    }
    Class<?> returnType = m.getReturnType();
    if(returnType.isInterface()){
      Class adaptedInterface = GenerateAdapterInterface(
         returnType);
      Class adapter = CreateAdapter(returnType,
         adaptedInterface);
      CreateAdapterAnnotation(returnType, adapter);
    }
  }

  /**
   * Creating a GlobalContext class
   */

  Class globalContext = GenerateGlobalContextClass(
     GetDataClasses(service));

  /**
   * Building the beans.xml file using the input parameters and
       the globalContext class
   */
  return BuildSOAPServerBeans(id, service, wsdl, endpointName,
     serviceName, url, globalContext);

}
```

The `deploy` method takes the following parameters: a service id, a service implementation class, a `WSDL` location, an endpoint name, a service name and the url of the Web service. First, the algorithm checks for interface types in the parameters or the return type

of the service methods in order to generate the corresponding adapters. Then, it generates a `GlobalContext` class based on all defined data classes. Finally, a beans file is created based on the deploy parameters and the built `GlobalContext` class.

At the client side, we propose to provide a new API which returns a proxy in accordance with an OO service required interface and a URL address. Figure E.1 presents an abstract UML diagram of the client API. The `getProxy` method in the `Client` class returns a proxy of type `T`, the type of the required interface given in the first parameter. This returned proxy is built from a generated `"beans.xml"` file using a `org.springframework.context.support.ClassPathXmlApplicationContext` class instance. The abstract code of the `getProxy` method is defined as follows:

```
<T> T getProxy(Class<T> c, String url){

  /**
  * Creating a GlobalContext class
  */
  Class globalContext = GenerateGlobalContextClass(
     GetDataClasses(c));
  /**
   * building the beans.xml file using the input parameters and
       the globalContext class
   */
  String beansLocation =  BuildSOAPClientBeans( c, url,
     globalContext);
  /**
   * create a context instance pointing on the beans file
       location
   */
  ClassPathXmlApplicationContext context
        = new ClassPathXmlApplicationContext(new String[] {
           beansLocation});

  return (T) context.getBean("client");
 }
```

Contrary to the server side, we do not check to adapt interfaces data type, because we consider, as it is standardized for `SOAP` services, that the client is built from the service provided `WSDL` file. Indeed, reproducing the interfaces types or not depends on the `java2wsdl` tool[5] (executed at the server) and `wsdl2java` tool[6] (executed at the

[5] see the `Java` first approach in the `cxf` user guide http://cxf.apache.org/docs/cxfeclipseplugininstructions

[6] see the `WSDL` first approach in the `cxf` user guide http://cxf.apache.org/docs/cxfeclipseplugininstructions

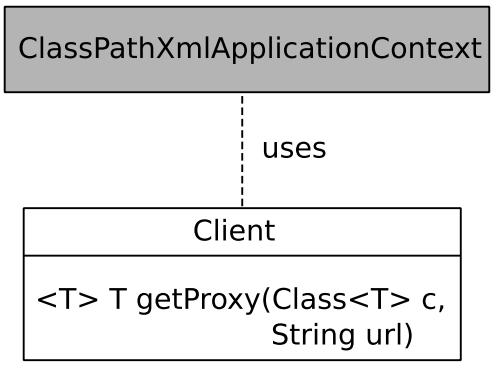


Figure E.1: An API to create a client by generating a beans.xml file

client) defined in `cxf`. Using the existing implementation of these tools, interfaces are not generated at the client side. These interfaces are replaced by their corresponding `AdaptedInterface` classes in the generated client code.

- `RESTful`: At the server side, at the deployment of the `Java` code as a `SOAP` service, the following abstract code should be applied:

```
File deploy(String id, Class service, String URL){
  /**
   * Creating an adapter when an interface is used as parameter
       type or return type in a service method
   */
  Method[] mList = anInterface.getMethods();
  for (Method m:mList){
    Class<?>[] paramTypes = m.getParameterTypes();
    for(Class<?> c : paramTypes){
      if(c.isInterface()){
        Class adaptedInterface = GenerateAdaptedInterface(c);
        Class adapter = CreateAdapter(c, adaptedInterface);
        CreateAdapterAnnotation(c, adapter);
      }
    }
    Class<?> returnType = m.getReturnType();
    if(returnType.isInterface()){
      Class adaptedInterface = GenerateAdapterInterface(
         returnType);
      Class adapter = CreateAdapter(returnType,
         adaptedInterface);
      CreateAdapterAnnotation(returnType, adapter);
    }
  }

  /**
```

```
    * Getting all data classes for the service
    */
  Class<?>[] dataClasses = GetDataClasses(service);

  /**
   * Building the beans.xml file using the input parameters and
       the globalContext class
   */
  return BuildRESTServerBeans(id, service, url, dataClasses);
}
```

The `deploy` method takes the following parameters: a service id, a service implementation class and the URL of the Web service. Contrary to the `SOAP` case, the `deploy` method does not mandatory require a structural interface as input parameter, for instance a `WADL` file like `WSDL` for `SOAP`. This is due to the fact that `WADL` is not yet a standard for `RESTful` like `WSDL` for `SOAP`. At execution, the `deploy` algorithm first checks for interface types in the parameters or the return type of the service methods in order to generate the corresponding adapters. Then, it gets all the data classes used by the service methods in order to use them as input parameter on `BuildRESTServerBeans` method, which creates the `"beans.xml"` file using also the parameters of the deploy method.

At the client side, we propose to provide a new API, like we have done for the `SOAP` case. This API has the same UML diagram as in Figure E.1 but a different implementation of `getProxy` method:

```
<T> T getProxy(Class<T> c, String url){
  /**
   * Creating an adapter when an interface is used as parameter
       type or return type in a service method
   */
  Method[] mList = anInterface.getMethods();
  for (Method m:mList){
    Class<?>[] paramTypes = m.getParameterTypes();
    for(Class<?> c : paramTypes){
      if(c.isInterface()){
        Class adaptedInterface = GenerateAdaptedInterface(c);
        Class adapter = CreateAdapter(c, adaptedInterface);
        CreateAdapterAnnotation(c, adapter);
      }
    }
    Class<?> returnType = m.getReturnType();
    if(returnType.isInterface()){
      Class adaptedInterface = GenerateAdapterInterface(
          returnType);
      Class adapter = CreateAdapter(returnType,
          adaptedInterface);
```

```
      CreateAdapterAnnotation(returnType, adapter);
    }
  }


  /**
   * Getting all data classes for the service
   */
  Class<?>[] dataClasses = GetDataClasses(service);

  /**
   * Building the beans.xml file using the input parameters and
       the service dataClasses
   */
  String beansLocation = BuildRESTClientBeans(id, service, url,
      dataClasses);

  /**
   * create a context instance pointing on the beans file
      location
   */
  ClassPathXmlApplicationContext context
        = new ClassPathXmlApplicationContext(new String[] {
           beansLocation});

  return (T) context.getBean("client");
 }
```

Contrary to the SOAP case, we check for adapting interfaces data type in the getProxy method, because the client code could be or not generated from a structural interface.

A real implementation of this automation algorithm is subject to future work.

E.5 MBeans configuration in cxf

cxf offers an MBeans instrumentation provided by a JMX implementation: "org.apache.cxf.management.jmx.InstrumentationManagerImpl"[7]. In order to integrate JMX to cxf using Spring XML on Tomcat, the following minimal XML snippet is required[8]:

```
<import resource="classpath:META-INF/cxf/cxf.xml"/>
...
```

[7] http://cxf.apache.org/javadoc/latest-2.4.x/org/apache/cxf/management/jmx/InstrumentationManagerImpl.html

[8] http://cxf.apache.org/docs/jmx-management.html

```
<bean id="org.apache.cxf.management.InstrumentationManager"
  class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
  <property name="enabled" value="true" />
  <property name="bus" ref="cxf" />
  <property name="usePlatformMBeanServer" value="true" />
</bean>
```

In order to get statistics for services running in `cxf`, `cxf` management module provides a feature (the `Performance.Counter.Server MBean`). To be able to see this information, the following configuration snippet is required in the Spring configuration file:

```
<!-- Wiring the counter repository -->
<bean id="CounterRepository" class="org.apache.cxf.management.
   counters.CounterRepository">
    <property name="bus" ref="cxf" />
</bean>
```

The `CounterRepository` collects the following metrics: invocations, checked application faults, unchecked application faults, runtime faults, logical runtime faults, total handling time, max handling time, and min handling time.

F

Résumé long en Français

Contents

Les services Web sont importants aujourd'hui car ils font partie de notre vie quotidienne : pour partager des photos en utilisant Flickr, pour acheter des produits à l'aide d'eBay ou pour payer en ligne en utilisant PayPal, nous utilisons les services Web. Les fournisseurs de ces applications offrent leurs services comme des interfaces (API). Ces interfaces sont accessibles par le biais de deux principaux protocoles (ou modèles) : `SOAP`, qui est un modèle basé sur des activités (style RPC) et `RESTful` qui est un modèle basé sur des ressources (style Web). La différence entre ces deux modèles peut être facilement représentée par l'exemple qui suit. On considère un service de réservation de vol : un client recherche un vol selon une ville d'origine, une ville de destination et une date, puis il réserve un vol. En `SOAP`, ce service est représenté par deux activités : "chercher un vol" et "réserver un vol". Ces deux activités sont accessibles via une URL, "http ://serviceDeReservationDeVol", (voir Figure F.1(a)). Dans `RESTful`, ce service est représenté par deux ressources : une ressource "Vol" et une ressource "Réservation". Chaque ressource a quatre opérations : get, put, post et delete. Pour chercher un vol, le client doit appeler l'opération get sur la ressource "Vol", tandis que pour réserver un vol, le client doit appeler l'opération put sur la ressource "Réservation". Chaque ressource est accessible par une URL spécifique qui est la continuation d'une URL de base de la ressource racine sous laquelle sont localisées les deux sous-ressources : "Vol" et "Réservation", (voir Figure F.1(b)).

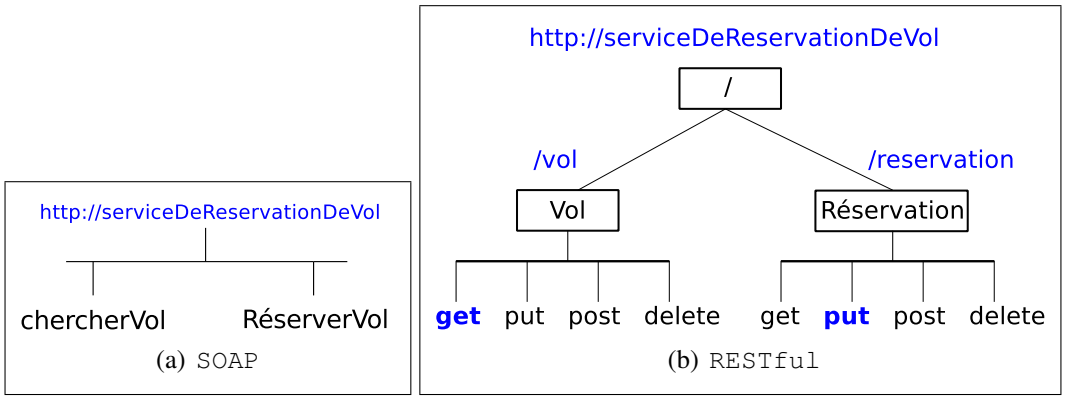


FIG. F.1 – Abstraction de service de réservation de vol en `SOAP` et `RESTful`

Actuellement, l'implémentation des services Web est surtout basée sur la programmation par objets pour deux raisons : (i) les langages objets sont connus de la plupart des développeurs, (ii) les services Web permettent couplage faible et interopérabilité. En utilisant un cadriciel, les développeurs peuvent transformer facilement un code à objet en un service Web, ou accéder à un service Web distant. Ces cadriciels sont principalement constitués de deux couches : une couche à objets (accessible directement aux développeurs) et une couche à services (cachée aux

développeurs). Ces deux couches ont chacune leurs propres caractéristiques et il est délicat de faire communiquer les deux couches en transportant les caractéristiques d'une couche à l'autre. Cette thèse se focalise sur cette problématique en se basant sur des jeux de test réels montrant des problèmes dans un cadriciel populaire pour le développement des services Web en `Java` : `cxf`.

Dans la suite nous présentons brièvement le contenu de cette thèse. Dans Section F.1, nous présentons l'architecture des cadriciels orientés objets pour la programmation des services Web. Section F.2 décrit les caractéristiques de chaque couche de ces cadriciels : la couche à objets et la couche à services. Nous mettons l'accent sur le principe de substitution de la couche à objets et le principe de découverte de la couche à services. Dans Section F.3, nous présentons des exemples de problèmes dans ces cadriciels. Dans Section F.4, nous présentons le besoin d'un modèle unifié pour la couche à services, il servira de fondement théorique pour améliorer ces cadriciels en respectant le sous-typage (imposé par le principe de substitution) et la découverte dynamique. Nous discuterons dans cette section quelques études de l'état de l'art qui peuvent nous être utiles. En Section F.5, nous présentons la formalisation du modèle unifié. Section F.6, décrit notre contribution pour transporter le principe de découverte dynamique dans la programmation à objet. Section F.7 détaille notre contribution pour transporter le principe de substitution dans la couche à services. Nous finissons dans Section F.8 par conclure ce travail en présentant quelques perspectives.

F.1 Architecture des cadriciels orientés objets

Avant d'entrer dans les détails de l'architecture des cadriciels orientés objets pour les services Web, nous commençons par un exemple introductif. Pour un service de réservation de vol, un développeur `Java` considère une interface `Java` définissant une opération, `reserver`, qui obtient en paramètre une instance de classe `Ticket` (pour des raisons de simplification, nous considérons **void** comme un type de retour de `reserver`). Pour déployer son code `Java` en tant qu'un service Web, le développeur peut convertir, en utilisant le cadriciel, l'interface `Java` en une interface standardisée structurelle : `WSDL` (pour `SOAP`) ou `WADL` (pour `RESTful`). Cette interface structurelle dépend d'un schéma où le type structurel de `Ticket` est défini. Un client qui souhaite appeler ce service, doit récupérer l'interface structurelle pour générer, en utilisant le cadriciel, l'interface `Java` et les classes correspondantes. La Figure F.2 illustre cet exemple.

Différents langage à objets et différents modèles de service existent et donc plusieurs cadriciels existent comme : `cxf`, RESTEasy, Restlet, Systinet, .Net, etc. Dans cette thèse, nous nous référons souvent au cadriciel `cxf` parce qu' : (i) il est un cadriciel populaire sous une licence Apache, (ii) il permet le développement de deux modèles de services, `RESTful` et `SOAP`, (iii) il utilise le langage `Java`, (iii) il s'agit d'une mise en oeuvre des normes (JAX-RS pour `RESTful` et JAX-WS `SOAP`). Comme le montre la Figure F.2, les cadriciels existants présentent deux couches : une couche à objets construite au dessus d'une couche à services.

Dans cette section, nous mettons l'accent sur les échanges entre ces deux couches afin de donner une vue abstraite de ces cadriciels. Nous nous concentrons sur un composant principal de ces cadriciels qui est responsable de la conversion des objets en structures et inversement, connu

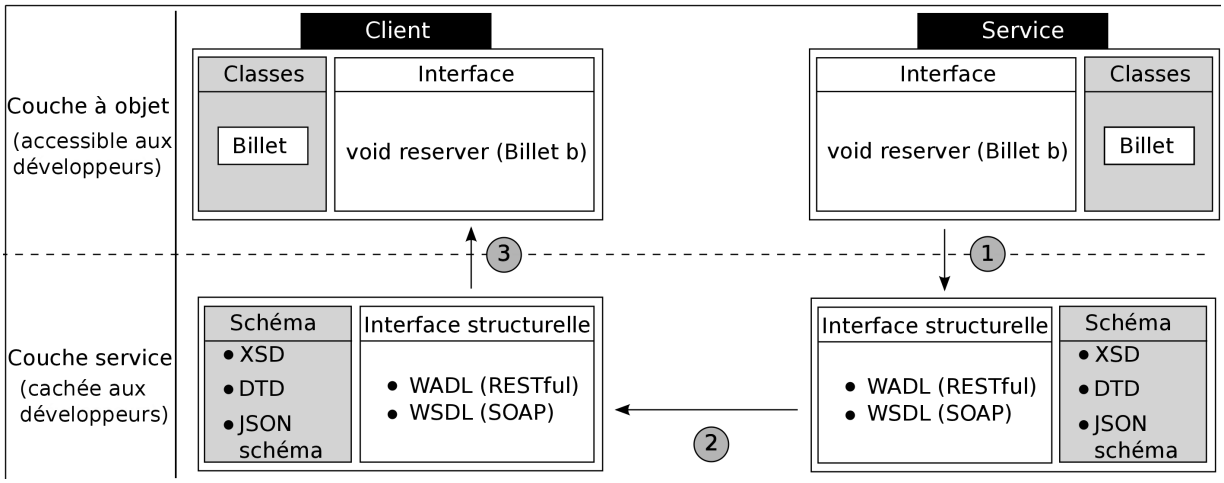


FIG. F.2 – Exemple de développement : réservation de vol

sous le nom de `Data Binder`. Ensuite, nous décrivons le flux de données échangé entre une source et une destination.

F.1.1 Data Binder

Le composant essentiel dans un cadriciel orienté objet comme `cxf` est le `Data Binder` [56], qui lie les types et les valeurs entre les deux couches. Dans la couche à objets, les valeurs sont des références à des objets et les types sont les types d'objets, des classes ou des interfaces. Dans la couche à services, les valeurs sont des documents structurés et les types sont des schémas. Concrètement, comme il y a différents langages pour les documents, les plus connus étant `XML` et `JSON`, un cadriciel comme `cxf` accepte différents `Data Binders` comme `JAXB` [43] (celui par défaut) et `Aegis` pour `XML`[1] ou `Jettison` et `Jackson` pour `JSON`[2].

Deux fonctions sur les types. Le `Data Binder` lie les types d'objets et les schémas représentant leurs structures internes dans une transformation bidirectionnelle. La `compilation de schéma` produit les types d'objets à partir d'un schéma tandis que la `génération de schéma` produit un schéma à partir des types d'objets. Par exemple, une classe `A` peut être liée à un type schéma, en donnant, pas seulement le nom du type, `A`, mais aussi sa structure comme une séquence de déclarations de champs. Dans Figure F.3, nous montrons un exemple d'une classe `A` liée à un schéma en donnant le nom du type structurel, en utilisant l'annotation `JAXB` `@XmlRootElement` (`name` = "A"), et la structure de type comme une séquence de déclarations de champs, en utilisant l'annotation `@XmlElement`.

[1] http://cxf.apache.org/docs/aegis-21.html
[2] http://cxf.apache.org/docs/json-support.html

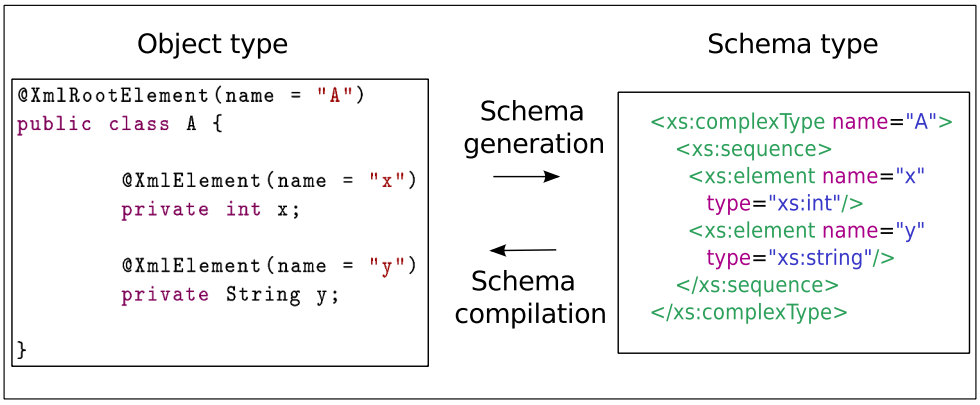


FIG. F.3 – Exemple de génération de schéma et de compilation de schéma

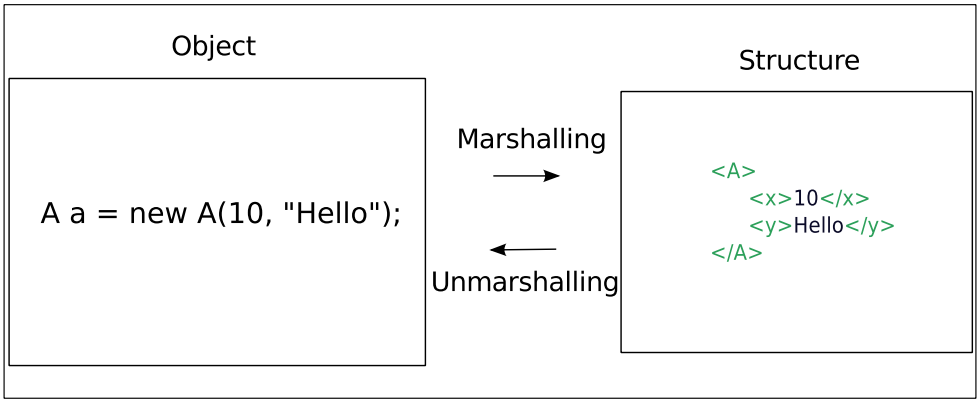


FIG. F.4 – Exemple de marshalling et d'unmarshalling

Deux fonctions sur les valeurs. En association avec les deux fonctions précédemment définies entre les types d'objets et les schémas, deux fonctions réalisent des conversions entre les objets et les documents. La fonction de `marshalling` convertie les objets en documents alors que la fonction d'`unmarshalling` convertie les documents en objets. Par exemple, une instance de la classe présentée précédemment `A` dans Figure F.3 est convertie en un document étiqueté avec le nom `A` et contenant une conversion de chaque champ comme un sous-document. La Figure F.4 montre le marshalling d'une instance de `A` suivi par l'opération inverse d'unmarshalling.

La génération de schéma, la compilation de schéma, le marshalling et l'unmarshalling décrits précédemment sont résumés dans Figure F.5.

Types Marshallables. La liaison de données est limitée à des types d'objets spécifiques, les `Types marshallables`. Un type d'objet est `marshallable` s'il satisfait certaines contraintes (sur ses constructeurs et champs) et définit une liaison spécifiques avec son schéma correspondant. Ces correspondances sont appelées `liaison de schéma` [56]. Elles sont décrites avec des annotations ajoutées au type d'objet comme dans `JAXB` ou avec des définitions

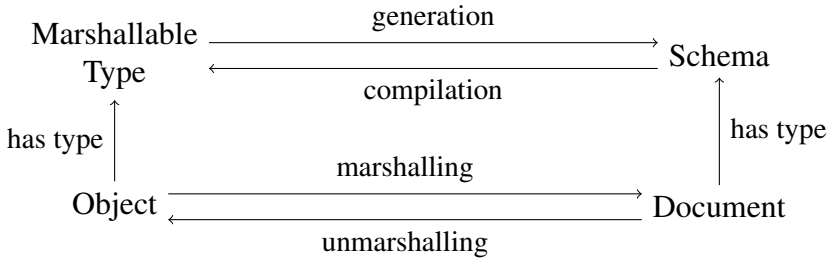


FIG. F.5 – Data Binding

dans un fichier séparé comme dans `Aegis`. Dans la suite, nous supposons implicitement qu'un type est marshallable.

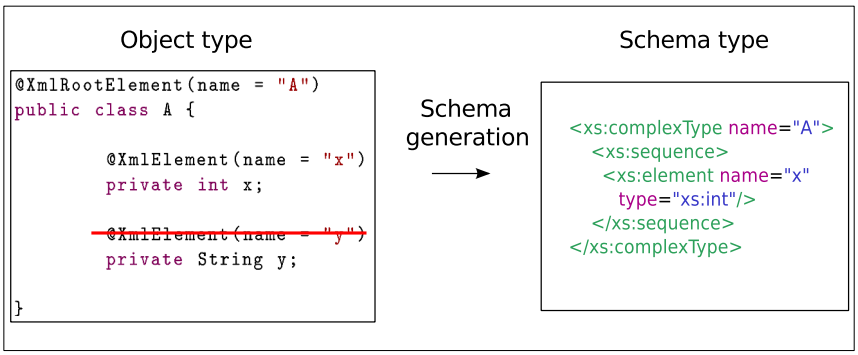
Équivalence des types. Les deux paires de fonctions, à l'échelle des types et des valeurs, respectivement, sont souvent présentées comme des paires de fonctions inverses. Formellement, ce n'est pas le cas. Tout d'abord, il y a un problème d'adaptation entre les schémas et les types d'objets [48], essentiellement parce que les langages utilisés pour définir des schémas sont trop expressifs. Mais même si nous nous limitons à des schémas générés par des types marshallables, il n'y a pas de correspondance biunivoque. En effet, étant donné un type marshallable, le schéma de liaison pourrait correspondre à certains attributs de la classe mais pas à tous. Par conséquence, la génération de schéma génère un schéma décrivant la structure des attributs spécifiée par la `liaison de schéma`, alors la compilation de schéma produit un type d'objet qui diffère de l'initiale : certains attributs peuvent manquer (voir Figure F.6).

La non-inversibilité vient du fait que la génération du schéma définit une procédure pour observer des objets, et cette observation est partielle : elle ne tient compte que d'une partie de l'état de l'objet observé et pas de toutes les méthodes encapsulées dans l'objet. De même, un marshalling suivi d'un unmarshalling ne conserve pas l'objet. Cependant, nous avons observé dans certains `Data Binders` que la propriété suivante est satisfaite, mais pas formellement indiquée : les différentes paires de fonctions sont quasi-inverses. Ainsi, à partir d'un schéma généré à partir d'un type marshallable, une compilation de schéma suivie par une génération de schéma conserve le schéma (voir Figure F.7(a)).

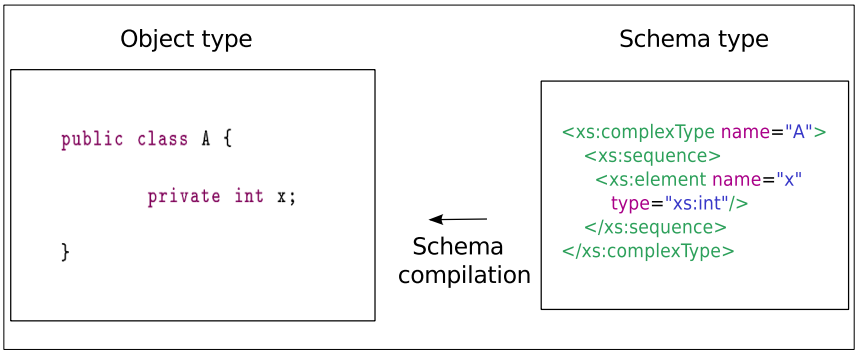
De même, dans le sens inverse, à partir d'un type d'objet compilé à partir d'un schéma, une génération de schéma suivie par une compilation de schéma préserve le type d'objet (voir Figure F.7(b)).

Ces propriétés induisent une notion d'équivalence spécifique sur les objets et les types d'objets respectivement : c'est la notion que nous allons utiliser dans la suite.

Definition 8 (Equivalence pour les types Marshallables) *Deux objets sont équivalents si l'application de la fonction de marshalling sur eux donne deux documents égaux, tandis que deux types marshallables sont équivalents si la génération de schéma appliquée sur eux donne deux schémas égaux.*

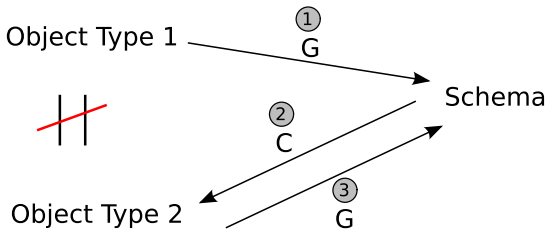


(a) Schéma génération

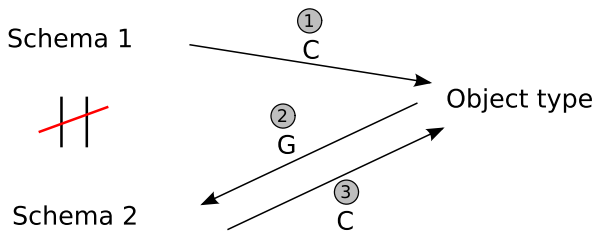


(b) Schéma compilation

FIG. F.6 – Exemple d'irréversibilité entre la génération de schéma et la compilation de schéma



(a)



(b)

FIG. F.7 – Quasi-reversibilité entre la génération de schéma (représentée par le symbole G) et la compilation de schéma (représentée par le symbole C)

Par exemple, avec JAXB, un type B est équivalent à son super-type A lorsque la classe B ne change pas la liaison de schéma héritée de A et ne l'étend pas par des champs supplémentaires.

F.1.2 Flux de données

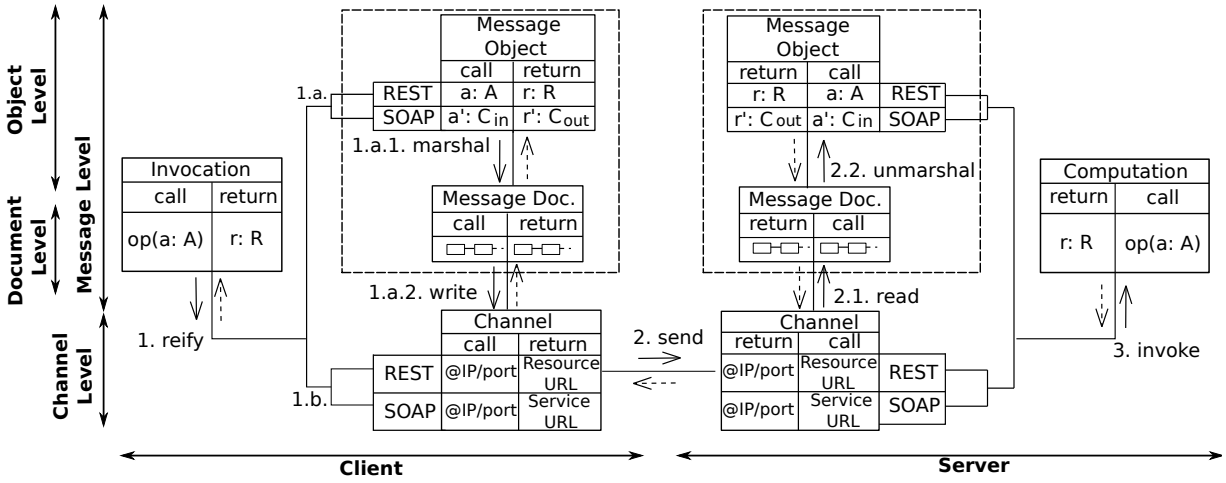


FIG. F.8 – Flux de données dans un cadriciel orienté objet pour les services Web

Le flux principal dans Figure F.8 est le marshalling du message objet en un message document, et l'unmarshalling correspondant. Le flux de données est essentiellement le même, pour SOAP et RESTful. La différence entre les deux technologies réside dans la façon dont une invocation d'une opération est réifiée en un message et un canal : la décomposition diffère. Dans ce qui suit, nous omettrons le flux de retour, car c'est la même analyse que pour le flux d'aller.

Cas SOAP. (i) Le message contient les arguments de l'opération, mais aussi une description de l'opération. Ainsi, comme représenté dans Figure F.8, le message objet résultant de l'appel op(a) est a', une instance d'un type C_{in} représentant les commandes associées aux appels à op et ayant comme attribut les paramètres d'entrée de cette opération.
(ii) Le canal identifie le port cible pour le service.

Cas RESTful. (i) Le message ne contient que les arguments de l'opération appelée. Ainsi, comme représenté dans Figure F.8, le message d'objet résultant de l'appel op (a) est tout simplement a, une instance du type d'entrée A.
(ii) Le canal identifie non seulement le service mais aussi l'opération comme une ressource et une méthode http.

F.2 Caractéristiques des couches à objets et à services

Chaque couche dans un cadriciel orienté objet a sa propre architecture conceptuelle : la couche à objets appartient à l'architecture pour objets distribués (AOD) [24] tandis que la couche à services appartient à l'architecture orientée services (AOS) [27]. Chaque architecture a ses

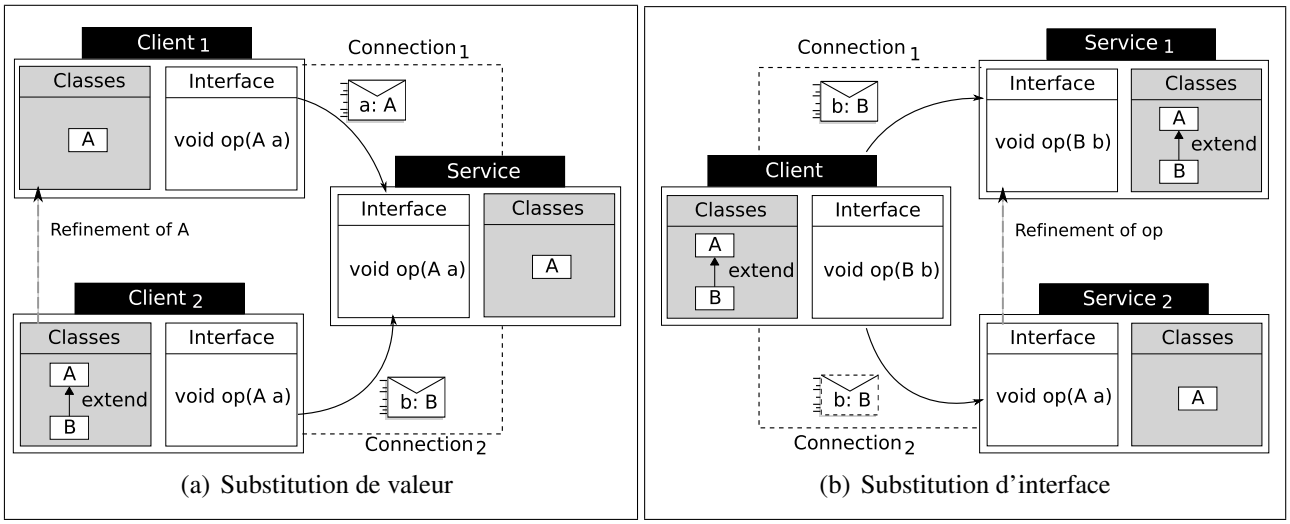


FIG. F.9 – Principe de substitution par l'exemple

propres principes. Dans la suite nous allons discuter en particulier le principe de substitution dans la couche à objets et la découverte dans la couche à services.

F.2.1 Principe de substitution dans la couche à objets

AOD soutient le *principe de substitution* [51] qui permet différentes substitutions dans la couche à objets : une valeur d'un sous-type pourrait être échangée entre le client et le serveur où une valeur d'un super-type est attendue. On distingue deux types de substitution :

Substitution de valeur. La Figure F.9(a) représente l'interface d'un service composé d'une opération (**void** `op(A a)`) et d'une hiérarchie de classes de données, du côté serveur et du côté client, avant et après un raffinement. Pour simplifier, nous supposons que la classe `A` est invariante après une génération de schéma suivie par une compilation de schéma afin de produire l'image de `A` du côté client. Après la génération du proxy côté client à partir du contrat déployé sur le serveur, la classe `A` est raffinée en une sous-classe `B`. En appliquant le principe de substitution, le client peut envoyer une instance de la classe `B` comme argument, à la place d'une instance de la classe `A`.

Substitution d'interface. La Figure F.9(b) représente deux services et un client. Le client est configuré initialement pour appeler $\texttt{Service}_1$. $\texttt{Service}_1$ est ensuite remplacé par un autre, $\texttt{Service}_2$, fournissant la même opération `op`, mais avec un sous type. Précisément, l'opération consomme un super-type `A` du type `B` tout en ne produisant rien. Par application de la règle de contravariance, le nouveau type raffine le premier. Encore une fois, pour simplifier, nous supposons que les classes `A` et `B` sont invariantes après une génération de schéma suivie par une compilation de schéma pour produire un proxy du côté client. En appliquant le principe de substitution, le client peut passer du $\texttt{Service}_1$ au $\texttt{Service}_2$ sans problème.

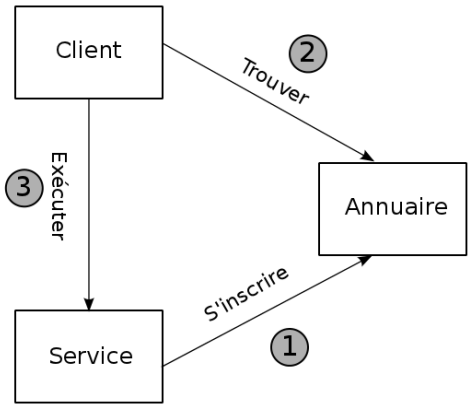


FIG. F.10 – Le triplet Client/Serveur/Annuaire dans l'architecture AOS

F.2.2 Principe de découverte dans la couche à services

AOS est basée sur un triplet, client/serveur/annuaire (voir Figure F.10) :
- les fournisseurs de services Web publient la disponibilité de leurs services,
- l'annuaire enregistre et classe les services publiés et fournit un service de recherche à ses clients,
- les clients cherchant un service utilisent l'annuaire afin de trouver un service pour l'utiliser ensuite.

Plusieurs technologies de recherche de services Web existent réellement : SOAP définit les standards UDDI [8] et WS-Discovery [1], tandis que RESTful définit Linked Data [37, 61].

La découverte peut être statique ou dynamique. Pour la découverte statique, le client interroge l'annuaire, au moment du développement, pour découvrir un service. Pour la découverte dynamique, deux méthodologies actuellement existent :
- Par mise à jour de la localisation d'un service : cette méthodologie est basée sur la substitution d'interface où l'interface ne change pas côté client mais seulement l'URL du fournisseur de service change,
- Par régénération d'interface : cette méthodologie est basée sur l'équivalence sémantique entre les services Web [85, 78, 84] où l'interface et l'URL du fournisseur de service changent côté client.

Dans cette thèse, nous nous intéressons uniquement à la découverte dynamique par mise à jour de la localisation du service.

F.3 Problèmes existants dans les cadriciels orientés objets

Après avoir présenté les principes fondamentaux qui caractérisent chacune des deux couches dans un cadriciel orienté objet pour le développement des services Web, nous nous posons la question suivante :

Connaissant les différences architecturales entre la couche à objets et la couche à services, comment est-ce que ces deux couches doivent être reliées entre elles tout en respectant les caractéristiques de chacune ?

Les cadriciels existants ont été construits d'une manière opérationnelle sans partir d'une spécification bien définie pour répondre à cette question. Ceci fait que les propriétés des deux couches ne se transportent généralement pas d'une couche à l'autre. Dans ce qui suit, nous présentons brièvement les trois principaux problèmes abordés dans cette thèse et qui sont dus à l'inadéquation entre ces deux architectures différentes.

F.3.1 Problème de sous-typage

Revenons sur l'exemple du service de réservation de vol, mais cette fois, nous considérons que le client ajoute certaines préférences sur sa demande pour réserver un billet. Le service peut considérer ces préférences ou pas. Prenons le cas lorsque le service ne traite pas ces préférences particulières mais traite toutes les demandes de la même manière. En terme de la programmation orientée objet, du côté client, un billet avec préférence peut être représenté par une instance d'une classe `PBillet` qui étend la classe `Billet`. Du côté serveur, seulement la classe `Billet` est connue. Le scénario est représenté dans Figure F.11. Cet exemple respecte le principe de substitution tel qu'il a été décrit précédemment.

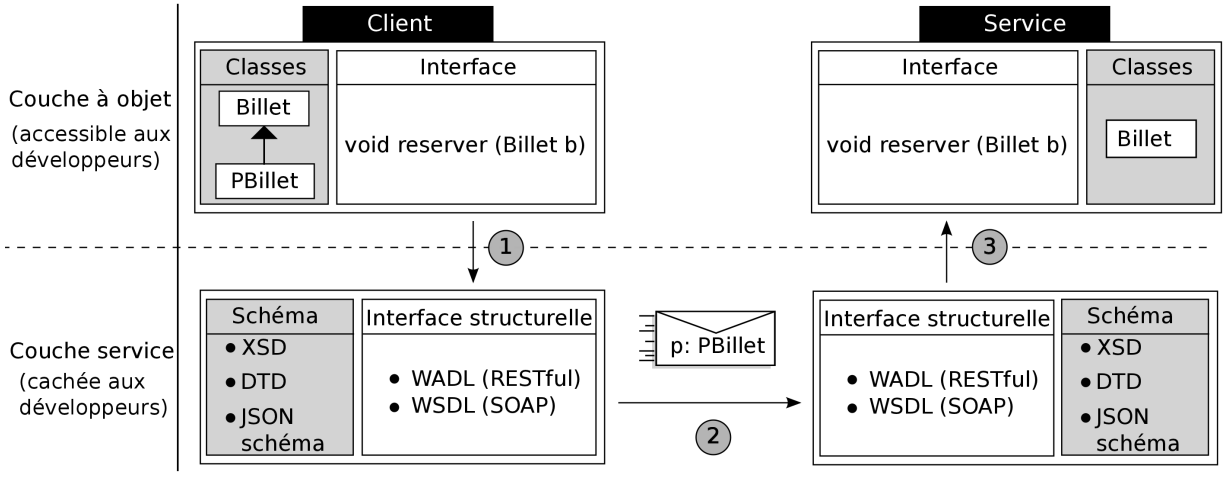


FIG. F.11 – Problème de sous-typage

En remplaçant `Billet` par `A` et `PBillet` par `B`, le scénario de la Figure F.11 peut être reproduit par le scénario de substitution de valeur et de substitution d'interface comme décrit précédemment dans Figure F.9. Dans la suite nous présentons les résultats de test de ce scénario en utilisant `cxf` suivant les deux types de substitutions.

Substitution de valeur. Différents résultats ont été observés selon la version de `cxf` : `2.5.x` ou `2.7.x`. Les résultats de tests sur `cxf 2.5.10` sont représentés dans Tableau F.12(a). Ce tableau montre certains résultats négatifs : la substitution fonctionne parfaitement

	SOAP	RESTful
Interopérabilité respectée	**Oui**	Non

(a) Substitution de valeur testé sur `cxf` 2.5.10

		SOAP	RESTful
Interopérabilité respectée	$B \equiv A$	**Oui**	Non
	$B \not\equiv A$	Non	Non

(b) Substitution d'interface

FIG. F.12 – Résultats de test du principe de substitution sur `cxf`

pour `SOAP` mais pas pour `RESTful`. Dans le cas `RESTful`, un `javax.xml.bind.UnmarshalException` est levé avec un message d'erreur indiquant que la structure du document reçu est inattendue car il a une étiquette `B` comme élément racine tandis que celle attendue devrait être `A`. Cependant, pour `cxf 2.7.5`, tous les résultats sont positifs à la fois pour `SOAP` et `RESTful`. Le guide de migration entre ces deux versions ne mentionne pas les modifications effectuées pour justifier ces résultats.

Substitution d'interface. Les résultats de tests entre le client et $\texttt{Service}_2$, présentés dans Tableau F.12(b), montrent certaines valeurs négatives : l'interopérabilité fonctionne partiellement pour `SOAP` et ne fonctionne pas du tout pour `RESTful`. Contrairement à l'exemple précédent, les résultats sont les mêmes pour toutes les versions de `cxf`. Dans le cas `SOAP`, lorsque `B` n'est pas équivalent à `A` (`B` définit un attribut en plus de `A` qui a une influence sur le type structurel correspondant, voir Définition 8), le serveur renvoie une exception de type `javax.xml.ws.soap.SOAPFaultException`. Ce message d'erreur est du à un élément supplémentaire inattendu dans le document reçu lorsqu'il est validé par rapport au schéma associé à la classe `A`. Dans le cas `RESTful`, les exceptions générées sont les mêmes que pour le scénario précédent pour la substitution de valeur.

Ces résultats montrent clairement l'échec du principe de substitution dans un cadriciel orienté objet pour les services Web comme `cxf`. On en déduit que les exigences souhaitables ne sont pas satisfaites.

- `Manque d'interopérabilité` **:** le client et le serveur sont plus fortement couplés qu'ils ne devraient l'être. En effet, l'interaction entre le client et le serveur ne peut pas bénéficier de la flexibilité induite par le principe de substitution.
- `Couplage fort` **:** la couche à objets et la couche à services sont fortement couplées dans `cxf`. En effet, il est impossible de migrer d'une technologie de service à une autre, par exemple du `SOAP` en `RESTful`, sans appliquer des adaptations sur le code côté client pour éviter les bugs.

La question qu'on se pause ici est :

Est-ce que ces erreurs détectées correspondent à des fautes ?

La lecture de la documentation, notamment le guide du développeur pour `cxf`, et de la spécification des normes mises en oeuvre, et l'instabilité du comportement que nous avons observée d'une version à une autre nous conduisent à la conclusion suivante : la validité du principe de substitution n'a pas été étudiée dans la spécification pour `cxf`. En d'autres termes, les erreurs détectées ne correspondent pas à des fautes, parce que le principe de substitution n'a pas été spécifié dans les besoins.

Cette inadéquation entre XML et la technologie orientée objet est un domaine bien connu [48], néanmoins des solutions pour combler les lacunes sont rares. L'approche dans [3] fournit un sous-ensemble de XSD et un algorithme pour convertir les documents XML en objets et vice-versa. C'est un point de vue qui part d'un document XML pour le convertir en un objet et de retourner ensuite à une représentation en XML. A notre connaissance, il n'y a pas d'études supportant le sens inverse : partant d'un objet vers un document XML pour ensuite retourner à une représentation objet plus particulièrement en considérant le sous-typage. C'est ce sens dont nous avons besoin pour résoudre le problème de substitution dans un cadriciel orienté objet pour les services Web.

Comme conclusion, nous avons alors besoin d'une spécification universelle incorporée dans le cadriciel et respectant le principe de substitution.

F.3.2 Découverte basée sur la substitution d'interface

Par rapport au scénario de substitution d'interface défini dans Figure F.9(b), la question est : comment est-ce que le client peut savoir que $Service_2$ est un sous-type de $Service_1$. Conformément au principe de découverte en AOS basé sur le triplet client/serveur/annuaire, le client doit s'adresser à un annuaire qui à son tour est capable de détecter $Service_2$ comme un sous-type de $Service_1$. La nécessité de la découverte avec sous-typage a été discutée par plusieurs auteurs [50, 47], cependant, le sous-typage est complètement absent dans les normes existantes pour la découverte comme UDDI et WS-Discovery.

F.3.3 Programmation de la découverte dans la couche à objets

Encore une fois, nous considérons le scénario de substitution d'interface défini dans Figure F.9(b), cette fois nous nous concentrons sur l'interaction entre le client et l'annuaire afin de découvrir en premier $Service_1$ ensuite $Service_2$. Le contexte est représenté dans Figure F.13.

Tout d'abord, le client demande à l'annuaire un service fournissant une interface I_1. L'annuaire envoie une référence à $Service_1$. Le client se connecte à $Service_1$. Dans un second temps, le client demande à nouveau à l'annuaire un service fournissant I_1. L'annuaire envoie une référence à $Service_2$. Le client effectue une connexion à $Service_2$. Dans cet exemple, nous supposons que $Service_1$ est un service RESTful et $Service_2$ est un service SOAP. Par conséquence, pour découvrir $Service_1$, le client doit utiliser le protocole de Linked Data, tandis que pour découvrir $Service_2$, un protocole comme UDDI est nécessaire. Ces deux normes de découverte ont différents degrés de difficulté et ces difficultés apparaissent dans la couche à objets en utilisant les API existantes pour ces normes. Par conséquence, le développement de la découverte dans la couche à objets devient une tâche complexe spécialement lorsque le protocole de découverte ou le modèle de service change, ainsi le code objet coté client doit subir des modifications profondes. Cependant, malgré les différences entre ces normes, il y a une procédure commune à respecter : chercher un service à l'aide de certains paramètres et ensuite récupérer une localisation d'un service correspondant. Il est possible d'unifier les concepts de la couche à services afin de rendre transparents les détails techniques de la découverte dans la couche à objets.

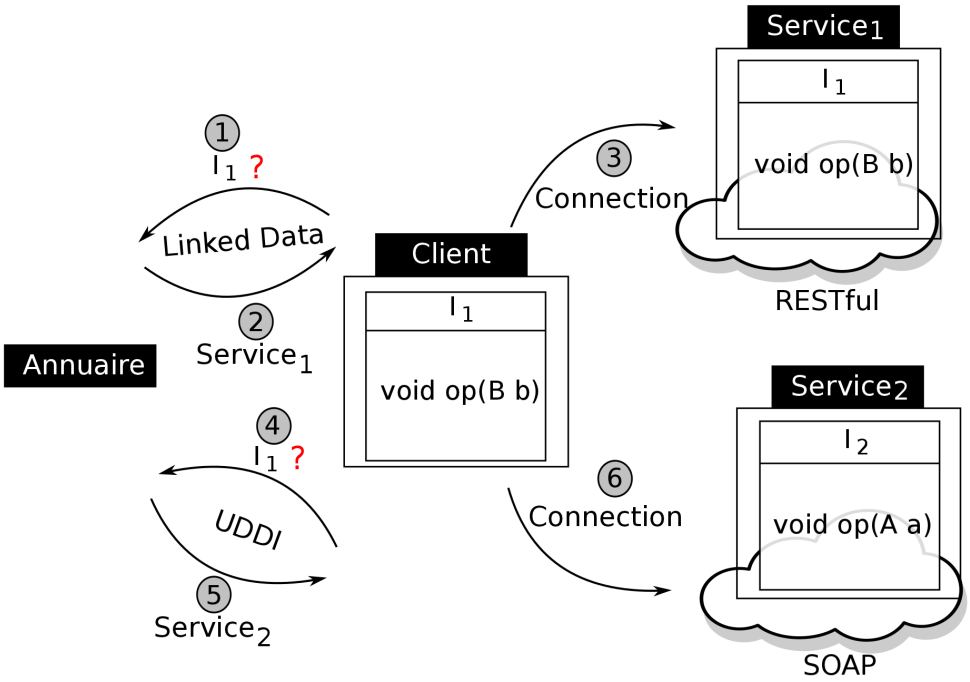


FIG. F.13 – Différentes APIs pour la découverte

F.4 Besoin d'un modèle unifié

Après avoir présenté les problèmes dans les cadriciels existants, nous pouvons déduire les besoins suivants :

- transporter les pratiques de découverte dynamique avec sous-typage dans les environnements de développement de services Web existants. Le moyen sera en facilitant et en unifiant l'accessibilité aux protocoles de découverte existants.
- définir une spécification pour permettre un développement orienté objet des services Web en considérant le sous-typage.

En combinant ces deux besoins, on en déduit que dans un environnement orienté objet pour le développement des services Web, il y a besoin d'un cadriciel qui :

- rend transparent les détails complexes de la couche à services afin de permettre la découverte des services Web d'une façon uniforme à la couche à objets indépendamment du modèle de service ou du protocole de découverte utilisé dans la couche à services,
- permet une interaction sûre en considérant le sous-typage sur les interfaces requises et fournies.

Pour atteindre cet objectif, il est nécessaire d'unifier les concepts à la couche à services afin d'offrir des facilités de développement dans la couche à objets.

Nous parlons d'un `modèle d'interaction`, ou encore un modèle unifié à `boîte noire` où un service est représenté comme un agent avec une interface structurelle (`WSDL` ou `WADL`). La Figure F.14(a) représente un tel modèle : le cercle noir recouvert d'un carré pointu est l'agent et l'arc est l'interface structurelle. Lorsque le service Web est mis en oeuvre à l'aide

(a) Interaction des services Web directement via des interfaces structurelles

(b) Interaction des services Web en utilisant une implémentation orientée objet

FIG. F.14 – Boites noires distribuées

d'un cadriciel orienté objet, l'agent est représenté par un cercle noir avec son interface objet, l'ensemble enveloppé par le carré pointu, comme il est représenté dans Figure F.14(b).

L'avantage d'un tel modèle à `boîte noire` dans les systèmes distribués est de dissocier l'agent de son interface :

- un service Web pourrait garder le même agent et modifier l'interface structurelle qui implique la modification du modèle du service Web,
- un service Web pourrait garder l'interface (le modèle de service Web) et changer l'implémentation de l'agent.

Ces deux propriétés impliquent deux points importants dans le contexte d'un cadriciel orienté objet de service Web :

- couplage faible entre la couche à services et la couche à objets
- l'interface objet et sa correspondante interface structurelle doivent être adaptées dans le sens que les messages sont échangés en toute sûreté à travers ces deux interfaces (voir Figure F.14(b)). Cette propriété implique que la relation de sous-typage doit être conservée du type objet en type service afin de préserver l'interopérabilité entre les agents distribués.

En conséquence, il est nécessaire de définir un modèle unifié typé pour `AOS` :

- premièrement, il est utile comme une formalisation de la couche à services avec un système de type expressif avec sous-typage,
- deuxième, il est utile pour montrer que les deux modèles de services Web, `RESTful` et `SOAP` pourraient être unifiés sur le plan conceptuel, malgré leurs différences,
- troisièmement, il contribue en tant que fondement théorique pour une API à objets pour le développement de la découverte dynamique (côté client), sur la base des concepts abstraits

unifiés,
– quatrièmement, le système de type du modèle unifié aidera à fixer les lacunes existantes entre l'interface à objets et l'interface structurelle correspondante dans le sens d'assurer l'interopérabilité par le sous-typage.

F.4.1 Modèle par envoi de message

D'après les critères de Lamport et Lynch [49], Les composants dans un modèle par envoi de message sont des boîtes noires, qui envoient et reçoivent des messages en utilisant un tampon (le réseau) sans partager une mémoire et sans la synchronisation de l'envoi et de la réception des messages dans un rendez-vous. Les modèles par envoi de message abstraient les détails de communication. Il existe quelques travaux intéressants sur le principe de boîte noire, comme [74, 44, 71]. Seehusen et Stolen dans [74] définissent un modèle formel et abstrait pour les services. La sémantique est basée sur une notion de trace qui est une séquence d'événements. Un événement est soit une transmission (!) soit une réception (?) de message composé (émetteur, récepteur, contenu). Le but de cette formalisation est de faire abstraction de séquence de message comme il l'est dans `UML 2`. Alors que dans [44], les auteurs utilisent la notion d'état abstrait pour préciser les descriptions fonctionnelles des services Web. Un service Web est défini comme une fonction totale qui lie les valeurs et les états aux traces d'exécution. Ces travaux ont des intérêts différents d'abstraction : [74] présente une formalisation avancée de certaines questions de confidentialité et [44] se concentre sur une description fonctionnelle particulière sans une syntaxe clairement définie.

Un modèle unifié pour `SOAP` et `RESTful`

Le problème d'intégration des deux modèles `SOAP` et `RESTful` a reçu une attention dans l'industrie[3]. Il existe peu d'études sur la composition et l'intégration des services hétérogènes dans le domaine académique. Une exception est [36] qui compare différentes techniques et propose une approche d'orchestration. Pratiquement leur solution est de concevoir des sous-workflows dédiés à la gestion de chaque technologie de manière native et à minimiser les interactions entre les deux parties. Le travail dans [6] aborde les questions d'interopérabilité dans le web, les services grid et p2p, mais il ne traite pas les services `RESTful`. Les auteurs préconisent un modèle conceptuel générique et abstrait basé sur les notions de service et de message. Une dernière approche consiste à fournir une transformation d'une technologie vers l'autre. C'est aussi le but de [76] qui définit les règles pour convertir automatiquement une architecture `SOAP` en `RESTful`.

En ce qui concerne les travaux sur les interfaces, Carpineti dans [17] fournit une notion de types de services qui inclut une notion de canaux typés et des opérateurs ensemblistes sur les types. Ce travail représente un fondement théorique pour les interfaces de services Web. Nous discuterons ce système de type dans F.4.2.

[3] `http://www-01.ibm.com/software/webservers/smash/#`, `http://ode.apache.org/`

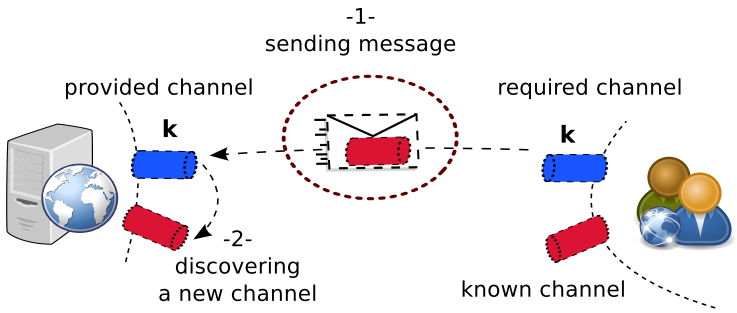


FIG. F.15 – Une communication client/server avec la découverte d'un canal

Du π-calcul au Join-clacul

Le π-calcul [63, 58, 70] est un calcul de processus classique qui est utilisé pour spécifier formellement et vérifier les systèmes concurrents. Il s'agit d'un modèle basé sur les interactions automatiques (Rendez-vous) et exige que l'expéditeur et le récepteur se synchronisent à des points spécifiques d'interactions, où ils peuvent échanger des informations de manière synchrone (il est toujours possible d'utiliser le π-calcul pour un modèle d'interactions asynchrone [16], mais il est moins connu pour une telle utilisation). Un système est représenté comme un ensemble d'agents ou de processus indépendants. L'interaction entre les processus est modélisée en utilisant des canaux. Un canal est une abstraction du média de communication sur lequel les données sont échangées. Le π-calcul permet également la transmission de canaux comme contenu dans les messages échangés à découvrir à la réception. Une illustration de ce principe est donnée dans Figure F.15, où nous représentons une communication entre un client et un serveur. Le client utilise le canal k fourni par le serveur. Le message envoyé sur k contient un canal qui est découvert à la réception par le serveur et ajouté à son interface. Cependant, le π-calcul est plus adapté aux orchestrations de services locaux que les systèmes distribués, parce qu'il manque la notion de localité sur les chaînes échangés [28]. Afin d'expliquer le principe de localité, nous prenons l'exemple d'un agent qui peut recevoir un canal, puis ajouter des règles sur ce canal. Par exemple, nous considérons un client, que nous appelons C, qui reçoit un canal l sur un canal d'entrée k, puis utilise l pour recevoir une donnée x avant exécuter P : $k(l).l(x).P$. Cet exemple est incompatible avec la mobilité et la distribution parce que le canal découvert l doit avoir un emplacement unique (URI) d'un agent spécifique que nous appelons p, de sorte que tous les messages envoyés sur l sont reçus par p et seulement par p. Par conséquence, l ne pouvait pas être utilisé comme un canal local pour recevoir des messages sur c. Pour résoudre ce problème, Fournet [28] propose une machine chimique réflexive associée au Join-calcul qui étend la machine abstraite chimique de Berry [11] (CHAM) avec la notion de localité et de réflexion :

– CHAM : elle apporte un comportement sémantique au π-calcul. Elle décrit le système comme une solution chimique, où les molécules interagissent entre elles et produisent de nouvelles molécules, selon des règles de réaction. D'autres règles, appelées règles parallèles, décomposent les molécules en molécules plus petites, ou composent des plus grandes molécules à partir des plus petites molécules. L'effet de ces règles, contrairement

aux règles de la réaction, est réversible.
- Localité : les molécules se déplacent directement à l'endroit où ils vont réagir et où le filtrage est appliqué uniquement sur le nom du canal. Chaque règle de réaction ou molécule peut être associée à un site de réaction unique.
- Réflexion : elle permet des réactions pour étendre la machine avec de nouveaux types de molécules avec leurs règles de réaction.

F.4.2 Système de type expressif avec sous-typage

Dans la suite nous présentons quelques besoins d'expressivité dans un modèle par envoi de message ensuite nous présentons brièvement le système de type de Castgna qui répond à nos besoins.

F.4.2.1 Besoin d'un type expressif

Nous avons besoin d'un système de type qui répond aux besoins suivants :

Type structure. Nous avons besoin d'un type qui permet de représenter la structure arborescente d'un document avec des étiquettes comme pour `XML` par exemple.

Type canal. Le type canal est utile pour exprimer la mobilité sous différentes formes :
- Demande/Réponse : dans un modèle de service asynchrone, le client doit communiquer un canal de réponse à son serveur lors d'une requête. A la réception, le serveur doit associer ce canal de réponse à son type de retour attendu (ce qu'on appelle une inférence de type). Ensuite, il va utiliser ce canal pour répondre au client. Ainsi, quand une réponse est attendue d'un service, un canal d'entrée au niveau du serveur doit définir un type qui contient un type de canal de retour utilisé pour déduire le type de la voie de réception par le client.
- Découverte de services : comme pour le principe de Demande/Réponse, la découverte de service nécessite la transmission des canaux. A la réception, l'agent découvre le nouveau canal et en déduit son type attendu. Ce canal découvert est utilisé pour appeler un service distant. Ainsi, un canal d'entrée, où il est prévu un canal de découverte, doit définir un type qui contient un type de canal de découverte utilisé pour déduire le type de canal reçu.

Récursivité des canaux. Parfois, on a besoin de représenter une chaîne d'interaction entre plusieurs serveurs qui route un message vers une destination finale. En plus, pour établir une connexion entre un client et un serveur, il est parfois nécessaire de mettre en place un certain nombre d'échanges valides. Ce type d'interactions pourrait être utile pour orchestrer des services (comme `BPEL` [60]) ou pour sécuriser des services (comme `OAuth` [35]).

Personnalisation des interfaces Web. Quand un fournisseur d'un service décrit son interface, il peut être amené à personnaliser les types d'entrée et de sortie. Afin d'enrichir le système de

type pour soutenir ces personnalisations, nous avons besoin de trois constructeurs : `Négation`, `Union` et `Intersection`. L'utilisation de ces constructeurs est utile pour traiter des données semi-structurées dans un langage de requête, `NoSQL` [9].

F.4.2.2 Système de type de Castagna

Pour répondre aux besoins d'expressivité de type précédemment présentés, nous avons besoin d'un système de type ensembliste. Le système de type de Castagna [29] étend un système de type avec des opérateurs ensemblistes :

$$t ::= \mathbf{0} \mid \mathbf{1} \mid t \to t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

0 et **1** correspondent respectivement aux types vides et universelles, $t \to t$ est le type de fonction, $t \times t$ est le type de produit, $\neg t$ est le type de négation, $t \vee t$ est le type d'union et $t \wedge t$ est le type d'intersection. En plus, ce système de type définit un algorithme de sous-typage (par inclusion).

F.5 Contribution 1 : modèle par envoi de message avec sous-typage

Afin de modéliser la communication entre les interfaces de services Web, le modèle chimique définie pour `Join-calcul` [28] correspond très bien. En outre, le système de type défini par Castagna répond à nos besoins pour la découverte dynamique et sous-typage. Cependant, ces deux travaux sont généraux en comparaison avec nos besoins. Le `Join-calcul` permet de modéliser des orchestrations locales (abstraites pour un modèle à boite noire) en plus du niveau de communication entre les services, comme prouvé par Fournet dans [28]. En outre, le système de type de Castagna est très général. Par conséquence, afin d'avoir un modèle simple plus dédié à l'interaction basée sur des interfaces Web, il est préférable de définir un modèle combinant la partie communication du `Join-calcul` et en utilisant le système de type de Castagna pour la découverte dynamique avec l'inférence de type.

Dans la suite, nous fournissons une formalisation d'un modèle de boîte noire pour la communication de services Web tout en étant abstrait par rapport aux détails et aux différences de technologie dans la couche à services. Ce modèle a deux caractéristiques principales :

- Découverte dynamique des canaux,
- Inférence de type des canaux découverts.

F.5.1 Modèle chimique de boite noire

Notre modèle appartient à la classe de `modèles par envoi de messages` [49]. Dans ce qui suit, nous décrivons brièvement le modèle formel. La sémantique opérationnelle de notre modèle est donnée par une machine chimique abstraite. Nous commençons par la partie syntaxique, qui décrit les composants. Puis, la mise en place de ces éléments produit un ensemble de particules dans un éther qui les contient.

Définition des composants

Environnement des services Web	ws	$::= \gamma$	Composant
		$\mid \gamma \parallel ws$	Composants en Parallèle
Composant	γ	$::= a[\sigma][I]$	Agent a avec un état σ et une interface I
Agent		$a \in \mathcal{A}$	Ensemble d'agents
Etat		$\sigma \in \Sigma$	Ensemble d'états
Interface	I	$::= \mathtt{0}$	Interface vide
		$\mid I \,\&\, c^{io}$	Interface composée de canaux
Canal	c^{io}	$::= c^{in}$	Canal d'entrée
		$\mid c^{out}$	Canal de sortie
	c^{in}	$::= k^{\iota}$	Canal d'entrée k
	c^{out}	$::= k^{o}$	Canal de sortie k
		$k \in \mathcal{K}$	Ensemble de canaux

TAB. F.1 – Composants et Agents

Nous supposons qu'un environnement de services Web est décrit comme un ensemble de composants en parallèle. Un composant est défini par un nom d'agent a, un état σ et une interface I. L'état des composants est maintenu abstrait, conformément avec le principe de boîte noire. Différents formalismes, comme les algèbres de processus, pourraient être utilisés pour modéliser des comportements des agents internes. Une interface peut être vide ($\mathtt{0}$) ou déclare (les noms) des canaux d'entrée et de sortie. Les canaux d'entrée k^{ι} correspondent aux canaux fournis par l'agent : les messages d'entrée sont reçus sur ces canaux. Les canaux de sortie k^{o} ont deux rôles différents : (i) ils correspondent à des chaînes d'émission de messages sur le réseau, (ii) ou ils peuvent être communiqués à un autre composant en les mettant dans le contenu du message pour être découvert à la réception. Le Tableau F.1 définit respectivement une représentation formelle des composants et des agents.

Déploiement des composants

L'environnement de services Web défini précédemment sera déployé dans une solution chimique que nous appelons "Ether". La particule peut se décomposer en des plus petites particules. Au cours de l'évolution de l'éther, les particules mobiles apparaissent : ils correspondent aux messages de sortie émis par les agents ($a[k^{o}(v)]$), les messages en transit (($k(v)$) et les messages d'entrée qui doivent être reçus par les agents ($a[k^{\iota}(v)]$).

Ether	Ω	$::= \langle\, \overrightarrow{\mu} \,\rangle$	Ensemble de particules μ
Particule	μ	$::= \gamma$	Composant
		$\mid a[\sigma]$	Agent avec état
		$\mid a[I^\iota]$	Agent avec une interface d'entrée
		$\mid a[I^o]$	Agent avec une interface de sortie
		$\mid a[m^{io}]$	Message local m^{io} chez l'agent a
		$\mid k(v)$	Message en transit
Message local	m^{io}	$::= m^{in}$	Message d'entrée
		$\mid m^{out}$	Message de sortie
	m^{in}	$::= k^\iota(v)$	Valeur d'entrée v sur le canal k
	m^{out}	$::= k^o(v)$	Valeur de sortie v sur le canal k

TAB. F.2 – Ether

Comme les différents messages peuvent avoir exactement la même forme (même canal, même contenu), l'éther est défini comme un ensemble $\langle\, \mu_1, \ldots, \mu_n \,\rangle$ de particules $\mu_1, \ldots, \mu_n$. Le Tableau F.2 résume formellement cette description. Dans ce tableau, nous notons par I^ι l'ensemble des voies d'entrée définies par un composant (l'interface d'entrée). De même, nous notons par I^o l'ensemble des canaux de sortie connus par un agent (l'interface de sortie). Nous supposons que $I^\iota \subseteq I^o$ car les canaux d'entrée peuvent être également transmis en tant que contenu dans les messages envoyés : chaque canal d'entrée est aussi une voie de sortie dans le sens où il peut être communiqué à un autre composant.

L'ether représente l'état global de l'ensemble de composants. Nous abstrayons le contenu des messages, à l'aide d'une simple structure, qui sera affiné dans la section suivante, avec un système de type

Trois règles de réduction sont nécessaires pour une interaction de services Web :

- La règle `[LOC]` : L'agent a consomme un ensemble, possiblement vide, de messages d'entrée m^{in}, et produit un autre ensemble, possiblement vide, de messages de sortie m^{out}, et met à jour son état de σ_1 à σ_2.
- La règle `[OUT]` : L'agent a envoie le message $k(v)$ sur le réseau. Une condition locale est indispensable : tous les canaux apparaissant dans le message ($\{k\} \cup \mathrm{K}(v)$) doivent être des canaux de sortie (I^o) déclarés par l'agent.
- La règle `[IN]` : L'agent a reçoit un message $k(v)$ du réseau. Une condition locale doit être satisfaite : le canal de message k doit être déclaré comme un canal d'entrée par l'agent. En outre, l'agent met à jour sa déclaration des canaux de sortie en ajoutant tous les canaux découverts dans le contenu v du message ($\mathrm{K}(v)$).

F.5.2 Système de type

Nous présentons dans cette section un système de type pour les valeurs portées par les services, *i.e.,* pour le modèle non typé présenté précédemment. Les valeurs sont construites selon les règles suivantes :

$$v \ ::= \ b \mid l[v], v \mid k$$

Une valeur v est soit une valeur primitive b ou un terme étiqueté $l[v], v$ (qui peuvent être utilisées pour construire des séquences de valeurs) ou un canal k. La syntaxe des types est la suivante :

$$t \ ::= \ \bot \mid \top \mid B \mid l[t], t \mid < t > \mid t + t \mid t \wedge t \mid \neg t \mid \mu X.t \mid X$$

Les types sont construits à partir d'un type de base B (désignant un ensemble de valeurs b), de constructeurs de valeurs ($l[\_], \_$) et un constructeur de canaux $< \ \_ \ >$. Ils peuvent être combinés en utilisant les opérations de réglage ($+, \wedge$ et $\neg$). Ils peuvent également utiliser la récursivité : les types récursifs peuvent être dépliés une infinité de fois, mais les valeurs sont limitées. Certains types récursifs ne sont pas constructifs (par exemple $\mu X.X$) ; nous considérons donc, comme d'habitude, seulement les types gardés : les constructeurs $l[\_]$ ou $< \ \_ \ >$ doivent se produire entre un liant quelconque μX et une occurrence de la variable X.

Suite aux travaux de Castagna [29], notre système de type peut être défini en termes d'un système de type sémantique en utilisant les opérateurs ensemblistes ($+$, $\wedge$ et $\neg$ sont les opérations standard). En effet, la grammaire de notre système de type est presque la même que dans [29]. Ils ont défini la syntaxe suivante pour les types :

$$t \ ::= \ \mathbf{0} \mid \mathbf{1} \mid t \rightarrow t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

Nous pouvons adapter notre syntaxe avec celui de Castagna :

- $\bot$ correspond à **0** et $\top$ à **1**,
- nous spécifions quelques types comme "base" pour les distinguer des types "étiquetés", alors qu'ils définissent un type général t sans faire de différence particulière entre les types,
- le type étiqueté $l[t], t$ est une limitation de l'utilisation de l'opérateur $\times$,
- le type canal, $< t >$ correspond à des fonctions de type $t \rightarrow t$ parce qu'un canal correspond à une fonction de service,
- les opérateurs $+$, $\wedge$ et $\neg$ correspondent à $\vee$, $\wedge$ et $\neg$,
- pour les types récursifs, nous les représentons avec des liants explicites $\mu x.t[x]$

Pour notre système de type, nous considérons l'algorithme de sous-typage défini dans [29]. Dans cette thèse, nous avons aussi montré la sûreté d'un tel système de type en utilisant les trois règles principales : `[LOC]`, `[IN]` et `[OUT]`. Nous avons appliqué la vérification de type, afin d'éviter les erreurs et les problèmes dus à la présence d'agents malveillants dans le réseau. Nous avons étudié deux niveaux de sûreté : *i)* tous les agents sont surs, *ii)* certains agents ne sont pas sûr.

F.6 Contribution 2 : transport de la découverte dynamique dans la programmation à objets

Différentes API orientées objets existent pour la découverte des services Web (modèles `SOAP` et `RESTful`). Ces API dépendent fortement des détails techniques de la couche à services ce qui rend leur utilisation très complexe. Malgré la différence qui existe entre ces APIs, la méthodologie de découverte est basée sur deux points essentiels :

1. l'interface du service à découvrir,
2. le protocole de découverte et la façon dont l'annuaire est appelé.

Dans cette thèse, nous présentons comment ces deux points pourraient être unifiés pour la découverte des services suivant les deux modèles `SOAP` et `RESTful` en utilisant notre modèle formel. Une fois que nous avons démontré la capacité d'unifier les différentes implémentations de découverte de la couche à services, nous sommes donc en mesure d'utiliser cette unification pour définir une API orientée objet unifiée et indépendante des détails techniques. Nous montrons comment cette API doit être construite de manière conforme aux pratiques de développement de la programmation par objets.

Dans la suite, premièrement nous présentons par un exemple comment les composants des services Web dans `SOAP` et `RESTful` peuvent être unifiés en utilisant notre modèle formel. Deuxièmement, nous montrons comment les différentes techniques de découverte des services Web pour `RESTful` et `SOAP` sont unifiées. Enfin, en se basant sur cette unification, nous présentons une API appropriée dans la couche à objets.

F.6.1 Unification des composants des services Web

Malgré la différence entre les deux technologies `SOAP` et `RESTful`, nous pouvons les unifier en utilisant notre modèle unifié présenté en Section F.5. Dans cette section, nous montrons comment les différentes notions de notre modèle correspondent avec les normes de `SOAP` et `RESTful`.

Nous illustrons notre approche par un exemple d'un système basé sur un service de réservation de vol. Nous considérons deux étapes : Dans une première étape, un composant client recherche d'abord un vol en spécifiant une ville d'origine, une ville de destination et une date. Dans une deuxième étape, il reçoit une liste de vols possibles, il fait son choix et il réserve son vol. Nous étudions deux implémentations différentes pour cet exemple, en utilisant respectivement `SOAP` et `RESTful`.

Conformément à notre modèle, le composant de service de réservation de vol est une composition d'une interface et d'un agent ayant un état interne qui évolue au cours de son exécution. L'abstraction de l'agent cache tous les détails de son implémentation. L'interface est composée de services fournis et/ou requis. Chaque service est un ensemble de canaux pour recevoir des messages entrants ou pour envoyer des messages sur le réseau.

Tout d'abord, nous formalisons le scénario de réservation de vol dans notre modèle formel. Le composant de service de vol fournit deux canaux $k_{searchTravel}$ et $k_{bookTravel}$ tel que c'est défini dans ce qui suit :

Composant du service de reservation de vol

$$
\begin{array}{lll}
\gamma_{service} & = & a_{service}[\sigma_0][I_{service}] \\
I_{service} & = & k^{\iota}_{searchTravel} : \texttt{< travelRequest, <travelReply> >} \\
 & & \ \& \ k^{\iota}_{bookTravel} : \texttt{< bookingRequest,} \\
 & & \qquad \texttt{<bookingConfirmation> >} \\
\texttt{travelRequest} & = & \texttt{request[source[string],} \\
 & & \texttt{destination[string],} \\
 & & \texttt{date[string], End], End} \\
\texttt{travelReply} & = & \texttt{reply[}\mu\texttt{X. (End + id[string], X], End)} \\
\texttt{bookingRequest} & = & \texttt{travel[id[string], End], End} \\
\texttt{bookingConfirmation} & = & \texttt{confirmation[bool], End}
\end{array}
$$

Le canal $k^{\iota}_{searchTravel}$ reçoit la demande de recherche d'un vol sous la forme d'un message contenant la ville de la source, la ville de destination et la date (une donnée de type $travelRequest$) et une voie de retour pour répondre à la demande de recherche avec l'ensemble des vols valides (données de type $travelReply$). Le canal $k^{\iota}_{bookTravel}$ reçoit un Id d'un vol (une donnée de type $bookingRequest$) afin de le réserver et un canal de réponse pour répondre à la demande de réservation avec une confirmation (une donnée de type $bookingConfirmation$).

Maintenant, considérons le composant client du service de réservation de vol. La formalisation de ce composant est représentée dans ce qui suit :

Composant client

$$
\begin{array}{lll}
\gamma_{client} & = & a_{client}[\sigma_0][I_{client}] \\
I_{client} & = & k^{o}_{searchTravel} : \texttt{< travelRequest, <travelReply> >} \ \& \\
 & & k^{\iota o}_{searchReply} : \texttt{<travelReply>} \ \& \\
 & & k^{o}_{bookTravel} : \texttt{< bookingRequest, <bookingReply> >} \ \& \\
 & & k^{\iota o}_{bookReply} : \texttt{<bookingConfirmation>}
\end{array}
$$

Nous ne détaillons pas ici les types définis car ce sont les mêmes que dans l'interface du fournisseur de service précédemment présenté. Les deux canaux fournis par le service sont requis par le client tel qu'il est représenté en utilisant l'annotation o. Deux autres canaux sont déclarés dans l'interface du client : $k^{\iota o}_{searchReply}$ utilisé pour recevoir la liste de vols et $k^{\iota o}_{bookReply}$ utilisé pour obtenir une confirmation de réservation. Ces deux canaux ont une double annotation ι et o : ils sont à la fois des canaux d'entrée et des canaux de sortie dans le sens que ces canaux peuvent sortir de l'agent en tant qu'un contenu dans un message à découvrir à la destination.

Dans ce qui suit, nous montrons la correspondance entre la formalisation précédente des deux composants (service/client) et des concepts concrets dans `SOAP` et `RESTful`. Pour chaque modèle, nous montrons comment une interface de service Web (`WSDL` ou `WADL`) pourrait être formalisée dans notre modèle formel.

- SOAP :

Pour le scénario de réservation de vol, nous supposons que l'adresse du port est L_{server} = "http ://flight-travel-service", le nom du Binding dans le WSDL est FlightTravelServiceBinding et l'interface de service définit deux opérations : searchTravel et bookTravel. L'opération searchTravel définit un message d'entrée travelRequest et un message de sortie travelReply. D'une manière cohérente, l'opération bookTravel définit un message d'entrée bookingRequest et un message de sortie bookingConfirmation.

Nous supposons que le client est accessible via un emplacement L_{client}, qui pourrait être :

- une adresse URL communiquée par le client au serveur en utilisant un entête ReplyTo dans le message SOAP au cas où la communication client/serveur est asynchrone (plus de deux sessions client/serveur distinguées),
- une adresse IP et un port sur http, dans le cas où la communication client/serveur est synchrone (généralement sur une seule session du protocole de transport).

Selon cette brève description de l'interface WSDL (pour une définition complète du fichier WSDL, voir Annexe D.1), nous associons les significations suivantes aux canaux précédemment présentés :

- $k_{searchTravel}$ est L_{server}.FlightTravelServiceBinding.searchTravel.searchRequest
- $k_{bookTravel}$ est L_{server}.FlightTravelServiceBinding.bookTravel.bookingRequest
- $k^{\iota o}_{searchReply}$ est L_{client}.FlightTravelServiceBinding.searchTravel.searchReply
- $k^{\iota o}_{bookReply}$ est L_{client}.FlightTravelServiceBinding.bookTravel.bookingReply

En ce qui concerne l'adéquation entre le type formel et les types déclarés dans le WSDL (voir la partie Type Definition du WSDL dans l'Annexe D.1), nous avons les correspondances suivantes :

- travelRequest = request[source[string], destination[string], date[string], End], End correspond à :

```
<element name="request">
  <complexType>
    <sequence>
      <element name="source" type="string"/>
      <element name="destination" type="string"/>
      <element name="date" type="string"/>
    </sequence>
  </complexType>
</element>
```

- travelReply = reply[μX. (End + id[string], X], End) correspond à :

```
<element name="reply">
  <complexType>
    <sequence>
      <element name="id" type="string" minOccurs = "0"
          maxOccurs = "unbounded"/>
```

```
      </sequence>
    </complexType>
  </element>
```

– `bookingRequest = travel[id[string], End], End` correspond à :

```
<element name="travelId" type = "string"/>
```

– `bookingConfirmation = confirmation[bool], End` correspond à :

```
  <element name="confirmation" type="boolean"/>
```

- `RESTful` : Le service de réservation de vol est composé de trois ressources :

– une ressource racine qui a l' `URL` racine du service : L_{root} ="http ://flightService/",

– une sous-ressource de la racine, `Vol`, accessible par l' `URL` : $L1_{server}$ = "http ://flightService/travel",

– une sous-ressource de la racine, `Réservation`, accessible par l' `URL` : $L2_{server}$ = "http ://flightService/booking".

La définition complète du fichier `WADL` pour le service de réservation de vol est présentée dans Annexe D.2. La méthode `GET` de la ressource `Vol` remplace la méthode `searchTravel` du `WSDL` de l'exemple précédent. La méthode `PUT` de la ressource `Réservation` remplace la méthode `bookTravel` définie dans le `WSDL`.

L_{client} a la même signification comme expliqué pour le cas `SOAP`. Nous associons la signification suivante pour nos canaux formels :

– $k_{searchTravel}$ est $L1_{server}$.GET.Request

– $k_{bookTravel}$ est $L2_{server}$.PUT.Request

– $k^{to}_{searchReply}$ est L_{client}.GET.Response

– $k^{to}_{bookReply}$ est L_{client}.PUT.Response

En ce qui concerne l'adéquation entre le type formel et les types dans le `WADL` (voir Annexe D.2), nous avons les correspondances suivantes :

– `travelRequest = request[source[string], destination[string], date[string], End], End` correspond à la liste de type `param` dans la partie `request` de la ressource `Vol` :

```
<request>
  <param type="string" style="query" name="source"/>
  <param type="string" style="query" name="destination"/>
  <param type="string" style="query" name="date"/>
</request>
```

– `travelReply = reply[`μ`X. (End + id[string], X], End)` correspond au type de `reply` dans la partie `grammars` du `WADL` :

```
<element name="reply">
```

```
    <complexType>
      <sequence>
        <element name="id" type="string" minOccurs = "0"
            maxOccurs = "unbounded"/>
      /sequence>
    </complexType>
  </element>
```

- `bookingRequest = travelId[string], End` correspond au type `param` dans la partie `request` de la ressource `Réservation` :

```
  <request>
    <param type="string" style="query" name="travelId"/>
  </request>
```

- `bookingConfirmation = confirmation[bool], End` correspond au type `reply` dans la partie `grammars` du `WADL` :

```
    <element name="confirmation" type="boolean"/>
```

F.6.2 Unification des protocoles de découverte dynamique des services Web

Dans cette section, nous donnons une formalisation abstraite d'un exemple de découverte dynamique appliquée au service de réservation de vol. Ensuite, nous présentons une abstraction globale qui préserve et unifie les protocoles existants pour la découverte. Nous avons déduit cette abstraction d'un ensemble de formalisations appliquées sur chacun des protocoles de découverte existants. Nous évitons d'entrer dans les détails de ces formalisations dans la suite, nous montrons juste l'abstraction globale que nous avons définie. Enfin, Nous montrons comment cette abstraction peut correspondre à notre modèle formel.

F.6.2.1 La découverte dynamique dans notre modèle formel

Supposons que le client ne connaît pas la localisation du service de réservation de vol mais connaît une interface requise de ce service. Ainsi, lors de l'exécution, avant d'invoquer le service, le client a besoin de découvrir un emplacement cible pour ce service. Un service est défini comme un ensemble de canaux correspondant à un port dans `SOAP` et à un ensemble de ressources dans `RESTful` (comme présenté précédemment). Alors la découverte d'un service implique la découverte d'un ensemble de canaux.

Pour illustrer le mécanisme de découverte de la formalisation dans le service de réservation de vol, le client a besoin de découvrir deux canaux `searchTravel` et `bookTravel` comme décrit précédemment. Au moment de la conception, le client est implémenté avec une interface requise pour un tel service : deux méthodes pour rechercher et réserver un vol. Au moment de l'exécution, pour invoquer ce service, le client demande à un annuaire une localisation du service en question. Nous supposons qu'il existe un composant qui fournit le service de réservation de

vol, γ_{server}. L'annuaire fournit un canal `searchFlightService` pour envoyer les canaux de γ_{server}. Un message entrant sur le canal `searchFlightService`, contient un canal de réponse sur lequel le client attend la réception des deux canaux :

- $searchTravel$ de type $< searchType >= \mathbf{<} travelRequest, \mathbf{<} travelReply \mathbf{>>}$,
- $bookTravel$ de type $< bookType >= \mathbf{<} bookingRequest, \mathbf{<} bookingReply \mathbf{>>}$.

Pour représenter le couple de canaux dans le canal de retour, nous définissons le type suivant :

$$reply :< service[fst[< searchType >], scd[< bookType >], End], End > .$$

Afin de simplifier la représentation de type, nous utilisons le sucre syntaxique suivante :

$$reply :<< searchType >, < bookType >>$$

$$reply : \mathbf{<<} travelRequest, \mathbf{<} travelReply \mathbf{>>}, \mathbf{<} bookingRequest, \mathbf{<} bookingReply \mathbf{>>} \mathbf{>>}.$$

La formalisation de l'ensemble des composants du système dans notre modèle formel est définie comme dans la suite (ici nous montrons que des détails concernant le client et l'annuaire) :

%%Global system:
$\gamma_{global} = \gamma_{server} \parallel \gamma_{client} \parallel \gamma_{registry}$

%%Individual Processes:
$\gamma_{client} = a_c[\sigma_{c0}][I_{client}]$
$\gamma_{registry} = a_r[\sigma_{r0}][I_{registry}]$

%%Interfaces description:
I_{client} = searchFlightService^{o}: <DiscoveryMsg>
&reply$^{\iota o}$: < <searchType>, <bookType> >
$=<$
$< travelRequest, < travelReply >>$
,
$< bookingRequest, < bookingReply >>$
$>$

$I_{registry}$ = searchFlightService$^{\iota}$: <DiscoveryMsg>
&searchTravel^{o}: <bookingRequest, <bookingReply> >
&bookTravel^{o}: <travelRequest, <travelReply> >

%%Discovery message type:
DiscoveryMsg $=<$
$< travelRequest, < travelReply >>$
,
$< bookingRequest, < bookingReply >>$
$>$

A l'exécution le client envoie le message :

$$m_1 = \texttt{searchFlightService}(\texttt{reply})$$

à l'annuaire. L'annuaire répond en envoyant les deux canaux du service de réservation de vol :

$$m_2 = \texttt{reply(searchTravel, bookTravel)}$$

Quand le client reçoit les deux canaux, il infère leur types et met à jour son interface. Cette formalisation est illustrée dans Figure F.16.

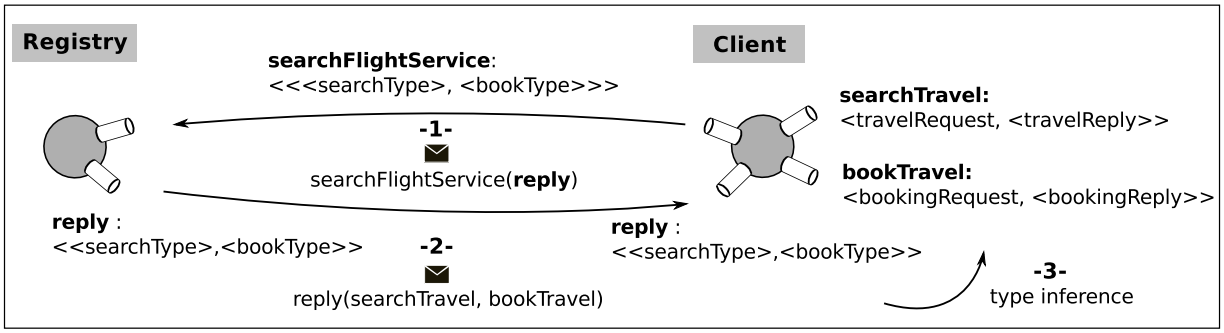


FIG. F.16 – Scénario de découverte

F.6.2.2 Abstraction général des protocoles de découvertes existants

La Figure F.17 montre une abstraction formelle des protocoles de découvertes existants. Le canal de découverte, $search$, fourni par l'annuaire a le type :

$$search^i : \forall\theta < Interface, < \theta >> .$$

$Interface$ est un type général représentant une information sur l'interface à découvrir et θ est un type générique faisant référence au type de l'interface du service à découvrir. θ est initialisée à l'exécution lorsque l'interface de service requise est connue. Du côté client, le canal de recherche sera utilisé pour découvrir le service de réservation de vol de type t_0, tel que t_0 est le type d'un couple de canaux représenté comme dans la suite (en utilisant le sucre syntaxique défini précédemment) :

$$< travelRequest, < travelReply > >, < bookingRequest, < bookingReply > >.$$

Le canal de découverte requis R est donc initialisé avec le type t_0 :

$$search^o_{t_0} :< Interface, < t_0 >> .$$

L'utilisation de type générique ne fait pas partie de notre syntaxe de type. Ainsi, la formalisation du mécanisme de découverte, tel que présentée dans Figure F.17, a besoin de quelques reformulations pour être typée dans notre modèle formel. Dans cette thèse, nous avons montré que la formalisation de la Figure F.17 est équivalente à celle de la Figure F.18 qui utilise l'opérateur de conjonction sur les canaux pour remplacer le type générique sur un ensemble fini de canaux connus par l'annuaire.

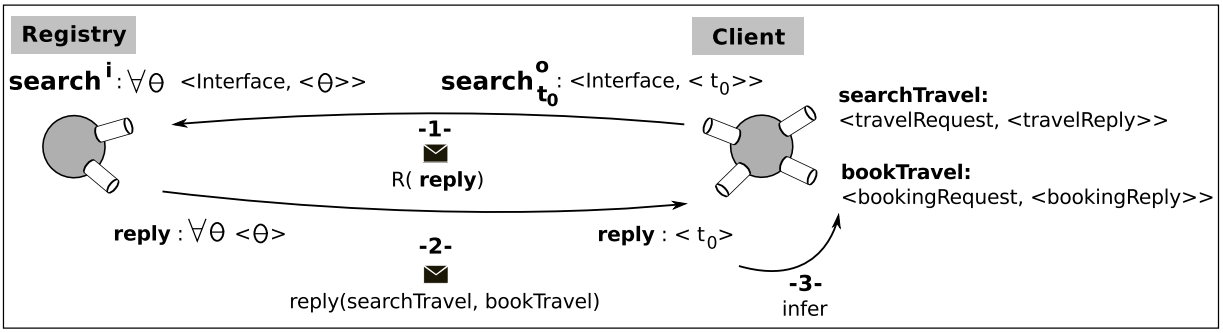


FIG. F.17 – Une abstraction de la découverte dynamique en utilisant le type générique

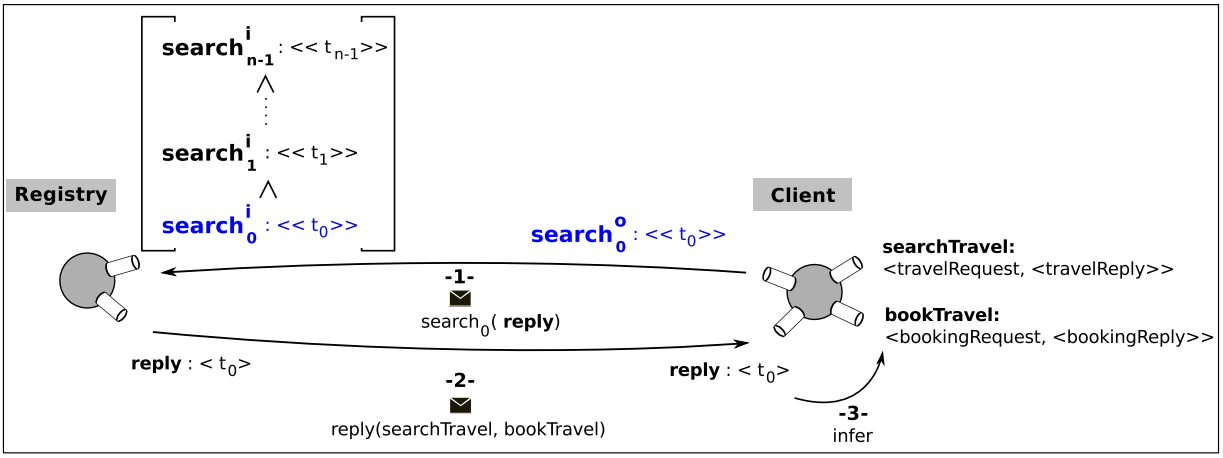


FIG. F.18 – Une unification de la découverte dynamique dans notre modèle unifié

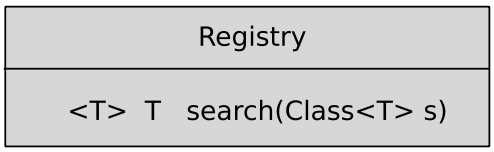


FIG. F.19 – Diagramme UML de l' API unifiée pour la découverte dynamique

F.6.3 Une nouvelle API orientée objet pour la découverte des services Web

Pour projeter l'unification précédente de découverte dynamique de services dans l'environnement à objets, nous avons le choix de considérer l'abstraction de la Figure F.17 ou de la Figure F.18. Comme les types génériques peuvent être représentées dans un langage à objets comme `Java`, nous choisissons l'abstraction de la Figure F.17. L'API orientée objet pour la découverte dynamique doit définir une opération `search` qui envoie une $Interface$ et reçoit un $Service$. Dans les termes de la programmation par objets, ceci correspond à envoyer une interface orientée objet et de recevoir une instance d'une implémentation de cette interface qui représentera un proxy local pour le service distant. L'opération de l'annuaire est définie comme dans la suite, (nous utilisons `Java` pour la démonstration) :

```
<T> T search(Class<T> interface);
```

Le diagramme UML de la classe de l'annuaire, `Registry`, est représenté dans Figure F.19.

En utilisant cette API, un développeur qui souhaite découvrir un service de réservation de vol (ayant `MyServiceInt` comme interface `Java`) pour appeler la méthode `searchTravel` a tout simplement besoin d'exécuter le code suivant :

```
public static void main(String args[]){
 // Instantiating a registry proxy
 Registry registryProxy = new Registry("http://registyURL");

 // Getting the service proxy
 MyServiceInt proxy =(MyServiceInt) registry.search(MyServiceInt.class);

 // Calling the searchTravel method
 TravelRequest req = new TravelRequest("Paris", "Berlin", "10/04/2014");
 List<TravelReply> reply = proxy.searchTravel(req);
}
```

Le développeur commence par créer une instance de la classe `Registry` en spécifiant l'`URL` d'un annuaire qu'il connaît. Puis, en appelant l'opération de recherche sur cette instance, il recevra un proxy du service de réservation de vol, une instance d'une implémentation de l'interface `MyServiceInt`. Sur le proxy du service obtenu, le développeur peut appeler directement toutes les opérations de l'interface `MyServiceInt`. Nous considérons la définition suivante de l'interface `MyServiceInt` :

```
 public interface MyServiceInt{
 public List<TravelReply> searchTravel(TravelRequest treq);
 public BookingReply bookTravel(BookingRequest breq)
```

```
}

public class TravelRequest{
  private String source;
  private String destination;
  private String date;
  // Getters and Setters
}

public class travelReply{
  private String id;
  // Getters and Setters
}

public class BookingRequest{
  private String travelId;
  // Getters and Setters
}

public class BookingReply{
  Boolean confirmation;
  // Getters and Setters
}
```

Dans cette thèse, nous avons montré comment cette API peut être développée au dessus des APIs de découverte existantes pour simplifier leurs usages. En plus, nous avons fourni une implémentation abstraite de cette API indépendamment du protocole de découverte utilisée par la couche à services et dans le but d'appliquer le sous-typage structurel sur les interfaces des services à découvrir chez l'annuaire.

F.7 Contribution 3 : transport du principe de substitution dans la couche à services

Dans cette section, nous présentons d'abord une abstraction représentant comment les cadriciels orientés objets contrôlent le `Data Binder` pour appeler ses différentes fonctions présentées dans Section F.1.1. Ensuite, en utilisant cette abstraction, nous montrons comment nous sommes en mesure de comprendre la cause des problèmes d'interopérabilité dans le cadriciel `cxf` dus au principe de substitution.

F.7.1 Contrôle du data binding

Dans Section F.1.1, nous avons présenté quatre fonctions essentielles sur les types et les valeurs : génération de schéma/compilation de schéma et marshalling/unmarshalling. Ici, nous montrons une formalisation de ces fonctions. Cette formalisation montre de façon abstraite comment un cadriciel orienté objet pour les services Web appelle ces fonctions.

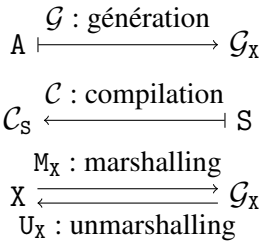


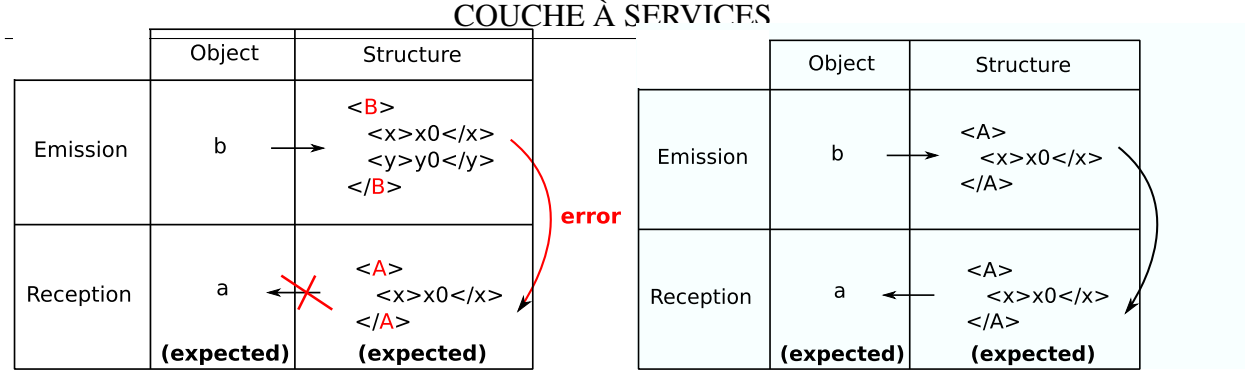
FIG. F.20 – Contrôle du `Data Binder`

Génération de schéma. Le cadriciel fournit un type objet et l'arbre associée de types liés. Le `Data Binder` renvoie un schéma représentant la structure des observations sur les types objets suivant la liaison de schéma définie. Le schéma de production est représenté dans Figure F.20 comme une fonction sur les types : la fonction $\mathcal{G}$ lie le type objet `X` au schéma $\mathcal{G}_X$.

Compilation de schéma. Le cadriciel fournit un schéma. Le `Data Binder` retourne un arbre de types objets. La compilation de schéma est représentée dans Figure F.20 comme une fonction sur les types : la fonction $\mathcal{C}$ lie le schéma `S` au type objet $\mathcal{C}_S$.

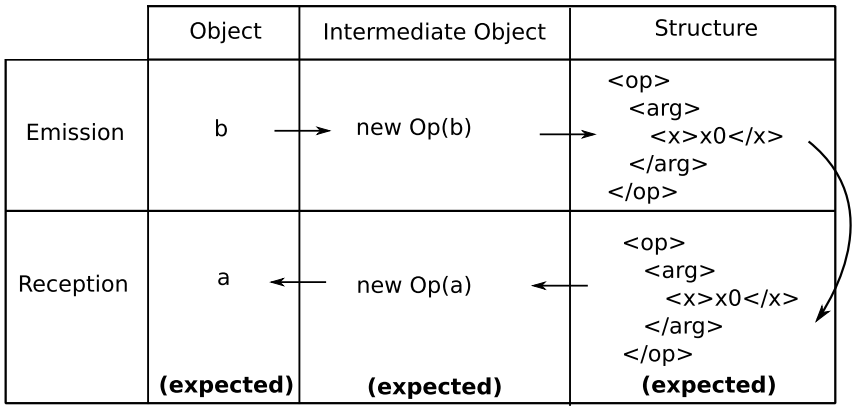
Marshalling. Le cadriciel fournit un objet. Le `Data Binder` renvoie un document représentant une observation de l'objet. Cette observation est basée sur les appels des accesseurs en lecture (getters) de sorte qu'elle produit une représentation, éventuellement partielle, de l'état de l'objet. La définition de la fonction de marshalling est récursive : l'observation de l'objet racine peut conduire à des objets qui doivent être observés, et ainsi de suite. Outre l'objet, un paramètre supplémentaire est nécessaire : en effet, la fonction de marshalling est une famille de fonctions, indexée par un type d'objet. La famille est compatible avec le sous-typage : l'observation d'un sous-type étend celle du supertype. Le type passé en argument par le cadriciel à la fonction de marshalling peut être le type dynamique de l'objet, déterminé dans `Java` par un appel de la méthode `getClass`, ou son type statique, déterminé à la déclaration de l'objet en question. Au marshalling, le cadriciel a le choix d'activer ou non la validation du document de sortie. Cette validation dépend du type structurel associé au type objet. La fonction de marshalling est une famille de fonctions M_X indexée sur l'objet de type `X` et définie à partir du type objet `X` vers le schéma $\mathcal{G}_X$, comme représenté dans Figure 8.1.

Unmarshalling. Le cadriciel fournit un document. Le `Data Binder` retourne un objet tel que son marshalling produit le document donné en entrée. En général, la construction de l'objet commence à partir d'un appel au constructeur de la classe d'objet sans argument, puis elle est basée sur les appels aux accesseurs en écriture (setters) associés aux getters utilisés dans l'observation pendant le marshalling. Le document fournit les arguments des setters. Là encore, la définition de la fonction d'unmarshalling est récursive : la construction d'un objet à partir d'une observation nécessite la construction des objets de sous-observations. Outre le document, un paramètre supplémentaire est nécessaire : un type objet utilisé pour construire l'objet retourné. Le type objet est déterminé statiquement, à la déclaration du service. A l'unmarshalling, le cadriciel a le choix d'activer ou pas la validation du document d'entrée. Cette validation dépend du type structurel associé au type objet. La fonction d'unmarshalling est notée une famille de fonctions



(a) `RESTful` : marshalling avec le type dynamique B à l'émission dans `cxf` 2.5.x

(b) `RESTful` : marshalling avec le type statique A à l'émission dans `cxf` 2.7.x



(c) `SOAP` : marshalling/unmarshalling avec le type statique A

FIG. F.21 – Les échanges entre l'émission et la réception pour le scénario de substitution de valeur

$\mathtt{U}_{\mathtt{X}}$ indexée sur l'objet de type `X` et définie du schéma $\mathcal{G}_{\mathtt{X}}$ vers le type objet `X`, comme représenté dans Figure F.20.

Pour conclure, nous pouvons maintenant formaliser la définition de l'équivalence entre les objets donnée de manière informelle dans Section F.1.1 (voir Définition 8).

Definition 9 (Equivalence pour les types marshallables) *Donnons deux types marshallables* `A` *et* `B`*, nous avons* `A` *est équivalent à* `B`*, et nous notons* $\mathtt{A} \equiv \mathtt{B}$*, quand* $\mathcal{G}_{\mathtt{A}} = \mathcal{G}_{\mathtt{B}}$*. Considérons* $\mathtt{A} \equiv \mathtt{B}$*; si l'objet* `a` *a le type* `A` *et l'objet* `b` *a le type* `B`*, nous considérons que* `a` *est équivalent à* `b` *quand* $\mathtt{M}_{\mathtt{A}}(\mathtt{a}) = \mathtt{M}_{\mathtt{B}}(\mathtt{b})$.

F.7.2 Scénario revisité : substitution de valeur

Basé sur l'abstraction du flux de contrôle précédemment présenté entre le cadriciel et le `Data Binder`, nous revisitons dans la suite le scénario de substitution de valeur comme présenté dans F.2.1 afin de localiser la cause des erreurs détectées. Nous ne revisitons pas ici le scénario de substitution d'interface comme les problèmes sont expliqués par une analyse similaire.

Grâce à l'évolution entre les versions étudiées, le diagnostic est facile. Dans le cas `RESTful` pour la version `2.5.10` de `cxf`, l'erreur provient du fait que le cadriciel appelle la fonction de marshalling à l'émission avec le type dynamique de l'objet (classe `B`), alors que le cadriciel appelle la fonction d'unmarshalling à la réception avec le type statique de l'objet (classe `A`). Comme montré dans Figure F.21(a), l'élément racine du document reçu, `b`, diffère de celui attendu, `a`. Ainsi, il n'est pas possible de réaliser l'unmarshalling en utilisant le type `A`. L'erreur est due à la non-équivalence entre les deux types (voir Définition 9). Dans le cas `RESTful` pour la version `2.7.5` de `cxf` (voir Figure F.21(b)) et dans le cas `SOAP` (voir Figure F.21(c)), le type statique, `A`, est utilisé pour le marshalling et l'unmarshalling. En d'autres termes, quand une instance de la sous-classe `B` est convertie comme une instance de la superclasse `A`, il y a aucune erreur.

Il était possible de forcer le sous-type, `B` et son supertype `A` à être équivalents en spécifiant la même liaison de schéma pour les deux types. Toutefois, la solution n'est pas universelle car elle implique une dépendance sur l'utilisation de la classe. Dans la prochaine section, nous montrons comment définir une spécification afin de satisfaire le principe de substitution. Le but est d'avoir une solution totalement transparente pour le développeur.

F.7.3 Une nouvelle spécification en utilisant les diagrammes commutatifs

Nous revisitons dans la suite le scénario de substitution de valeur tout en le généralisant pour proposer de nouvelles exigences sur le `Data Binder` : l'objectif est d'assurer la validité du principe de substitution.

Pour exprimer les exigences, nous utilisons principalement des diagrammes, qui sont des graphes avec des sommets représentant des types et des flèches représentant des fonctions. Ils représentent une abstraction de flux de données décrite dans Section F.1.2 : une exécution est décrite en tant qu'un chemin entre types, correspondant à une séquence de transformations de données appartenant à ces types. Ils permettent à l'application de la règle de subsomption d'être représentée comme une fonction de conversion. Par exemple, pour les types d'objet `A` et `B`, avec `B` sous-type de `A`, si une instance de `B` est convertie en `A` au cours de l'exécution, alors nous pouvons représenter la conversion par le diagramme suivant :

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

où `i` est la fonction de conversion canonique, définie dans `Java` comme `A i(B x) {return x;}`. Dans de tels diagrammes, une propriété de commutativité est nécessaire : tous les chemins ayant la même source et la même cible sont égaux. Les exigences sont divisées en `Exigences de base` qui formalisent les exigences satisfaites déjà dans les `Data Binders`, et `Nouvelles exigences` qui impliquent le sous-typage et permettent la validité du principe de substitution. Les nouvelles exigences que nous proposons ont deux objectifs concrets :

– éviter tous les problèmes détectés,
– assurer que tout objet à l'émission est équivalent à l'objet correspondant à la réception.

Dans un schéma, une erreur est représentée comme une flèche pointillée.

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

Dans la suite, nous allons détailler le cas `RESTful` du scénario pour la substitution de valeur. Nous présentons ensuite brièvement notre solution pour résoudre les problèmes existants pour `RESTful` et `SOAP` comme présentés dans Section F.3.1.

Considérons la situation initiale du scénario, lorsqu'un client envoie une instance de type `A` au serveur fournissant une opération **void** `op (A a)` : pas de sous-typage dans ce cas. Les processus de développement et d'exécution peuvent être représentés comme suit.

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

Le processus de développement permet la construction d'une séquence de types, de droite à gauche : appliquée à `A`, la génération de schéma produit le schéma $\mathcal{G}_{\mathtt{A}}$ et alors la compilation de schéma produit le type objet $\mathcal{CG}_{\mathtt{A}}$. A noter qu'ici et dans la suite, nous considérons le cas général, quand les types objets `A` et $\mathcal{CG}_{\mathtt{A}}$ ne sont pas supposés être égaux : dans F.3.1, pour simplifier, nous avons supposé qu'ils étaient égaux. Le processus d'exécution permet à la séquence de transformations (flèches) d'être construite, de gauche à droite. La fonction de marshalling $\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}$ transforme une instance de type $\mathcal{CG}_{\mathtt{A}}$ en un document conforme au schéma $\mathcal{G}_{\mathtt{A}}$ pour la transmission. Après la réception, la fonction d'unmarshaling $\mathtt{U}_{\mathtt{A}}$ transforme un document en un objet de type `A`. Comme la fonction de marshalling $\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}$ a le type $\mathcal{CG}_{\mathtt{A}} \rightarrow \mathcal{GCG}_{\mathtt{A}}$, nous avons besoin de la propriété suivante pour assurer que la transformation est bien typée.

Exigence de base 1 (Inversibilité de compilation) *La génération de schéma est une rétraction (inverse à gauche) de la compilation de schéma, quand on se limite aux schémas issus de la génération de schéma.*

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

L'équivalence entre l'objet initial et l'objet final peut être représentée par le schéma suivant, en considérant qu'il est commutatif.

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

En d'autres termes, les objets sont équivalents si leurs marshallings sont égaux, ce qui conduit à la propriété suffisante suivante, complétant l'exigence précédente.

Exigence de base 2 (Inversibilité d'unmarshalling) *La fonction de marshalling est une rétraction de la fonction d'unmarshalling.*

$$\forall\, \mathtt{A} \,.\, \mathtt{U}_{\mathtt{A}} \,;\, \mathtt{M}_{\mathtt{A}} = \mathrm{id}_{\mathcal{G}_{\mathtt{A}}}.$$

Nous considérons maintenant les exigences portantes sur le sous-typage. Nos schémas sont enrichis avec une autre ligne : la ligne du bas traite les sous-types tandis que la ligne du haut traite les supertypes.

Retournons à notre scénario de substitution de valeur dans sa forme simplifiée : un client envoie une instance de sous-type `B`, tandis que le serveur attend une instance de type `A`. Après la généralisation, nous obtenons les diagrammes suivants, correspondant pour le côté gauche à l'ancienne version de `cxf` et pour le côté droite à celle plus récente.

$$\begin{array}{ccc} \mathcal{G}_{\mathtt{A}} & \xrightarrow{\mathtt{U_A}} & \mathtt{A} \\ \vdots & & \\ \mathtt{B} & \xrightarrow[\mathtt{M_B}]{} & \mathcal{G}_{\mathtt{B}} \end{array} \qquad \begin{array}{ccccc} \mathcal{CG}_{\mathtt{A}} & \xrightarrow{\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}} & \mathcal{G}_{\mathtt{A}} & \xrightarrow{\mathtt{U_A}} & \mathtt{A} \\ \uparrow \mathtt{i} & & & & \\ \mathtt{B} & & & & \end{array}$$

Le type d'objet `B` est un sous-type de $\mathcal{CG}_{\mathtt{A}}$, comme représenté par la fonction de conversion `i` dans le schéma de droite. Le diagramme de gauche montre une erreur : le cadriciel appelle la fonction de marshalling avec le type dynamique `B`, produisant un document du schéma $\mathcal{G}_{\mathtt{B}}$; ce document ne peut pas être converti en une valeur du schéma $\mathcal{G}_{\mathtt{A}}$. D'autre part, le schéma de droite ne montre aucune erreur : le cadriciel appelle la fonction de marshalling avec le type statique, $\mathcal{CG}_{\mathtt{A}}$. Pour éviter l'erreur, il suffit de prévoir la possibilité de convertir les schémas, ce qui conduit à l'exigence suivante.

Nouvelle exigence 8 (`Lifting` de la génération de schéma) *La génération de schéma $\mathcal{G}$ est un foncteur* [4] *et la fonction de marshalling* `M` *est une transformation naturelle : pour tout type objet* `A` *et* `B` *et pour toute fonction $f : \mathtt{B} \to \mathtt{A}$, il existe une fonction $\mathcal{G}_f : \mathcal{G}_{\mathtt{B}} \to \mathcal{G}_{\mathtt{A}}$,* `lifting` *de f, de telle sorte que le diagramme suivant devient commutatif :*

$$\begin{array}{ccc} \mathtt{A} & \xrightarrow{\mathtt{M_A}} & \mathcal{G}_{\mathtt{A}} \\ f \uparrow & & \uparrow \mathcal{G}_f \\ \mathtt{B} & \xrightarrow[\mathtt{M_B}]{} & \mathcal{G}_{\mathtt{B}} \end{array}$$

Le `lifting` $\mathcal{G}_f : \mathcal{G}_{\mathtt{B}} \to \mathcal{G}_{\mathtt{A}}$ est commutable grâce à Exigence de base 2 et à Nouvelle exigence 8 :

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

De plus, avec la nouvelle exigence, le choix entre un type statique ou dynamique, pour passer à la fonction de marshalling, n'a pas d'importance. En effet, le diagramme suivant est devenu commutatif (grâce à Exigence 1 qui donne l'égalité $\mathcal{GCG}_{\mathtt{A}} = \mathcal{G}_{\mathtt{A}}$).

[4] Une fonction définie sur les types est un `foncteur` quand elle peut être étendu à une fonction par des fonctions typées.

$$\begin{array}{ccccc} \mathcal{CG}_{\mathtt{A}} & \xrightarrow{\mathtt{M}_{\mathcal{CG}_{\mathtt{A}}}} & \mathcal{G}_{\mathtt{A}} & \xrightarrow{\mathtt{U}_{\mathtt{A}}} & \mathtt{A} \\ \mathtt{i}\uparrow & & \uparrow\mathcal{G}_{\mathtt{i}} & & \\ \mathtt{B} & \xrightarrow[\mathtt{M}_{\mathtt{B}}]{} & \mathcal{G}_{\mathtt{B}} & & \end{array}$$

Résumé de résultat. En répétant la même analyse sur les deux scénarios de substitution de valeur et d'interface, nous avons conclu que tous les problèmes présentés dans nos scénarios étudiés proviennent d'une erreur de calcul au niveau de la fonction de `lifting`, utilisée pour la conversion d'objets en documents. Dans ce qui suit, nous présentons le bon calcul de cette fonction.

– Cas `RESTful` : si `i` est la conversion canonique de `B` en `A`, le lifting pour le cas `RESTful` est :

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

– Cas `SOAP` : pour le cas `SOAP`, la séquence de transformation du client vers le serveur est complexe durant un appel $\mathtt{op}(\mathtt{a})$ parce qu'il s'agit d'une réification de l'appel, comme présenté dans Figure F.8. En effet, à l'émission, avant le marshalling, l'argument, un objet `a` de type `A`, est d'abord plongé dans une `commande` qui est de type $\mathtt{C}_{\mathtt{in}}$ et réifie l'appel. Puis c'est la commande qui est convertie et envoyée. Symétriquement, à la réception, l'unmarshalling produit une commande `c` de type $\mathtt{C}_{\mathtt{in}}$. Une projection est ensuite appliquée sur la commande `c`, pour extraire l'argument. Pour décrire l'appel de réification, nous utilisons une fonction sur les types, correspondante à la génération de commandes, et deux transformations, indexées sur les types objets, des types objets dans des types commandes et inversement, un plongement et une projection respectivement.
  – Génération de commande : $\mathcal{F}_{\mathtt{A}}$ représente le type de commande associé au type objet `A`. Nous omettons la dépendance sur l'opération, qui est supposée fixe.
  – Plongement : Compte tenu d'un type objet `A`, $\mathtt{E}_{\mathtt{A}}$ représente la fonction de plongement définie de `A` dans $\mathcal{F}_{\mathtt{A}}$.
  – Projection : Étant donné un type objet `A`, $\mathtt{P}_{\mathtt{A}}$ représente la fonction de projection définie de $\mathcal{F}_{\mathtt{A}}$ à `A`.

  Figure F.22 résume ces définitions.

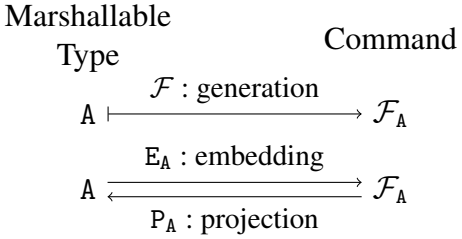


FIG. F.22 – Génération de schéma

Si i est la conversion canonique de B en A, le lifting pour le cas SOAP est :

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

F.7.4 Une concrétisation de la spécification

Nous présentons maintenant la concrétisation des fonctions de lifting, suivant Equation F.1 et Equation F.2. Notre objectif est de conclure une spécification qui doit être respectée dans chaque cadriciel orienté objet afin de permettre l'interopérabilité par sous-typage. Cette solution devrait être intégrée dans le cadriciel d'une manière transparente pour les utilisateurs.

La signification de la spécification. La spécification donnée par Equation F.1 et Equation F.2 peut être interprétée comme suit. Pour le cas RESTful, sur l'entrée $\mathtt{doc}_\mathtt{B}$ de type $\mathcal{G}_\mathtt{B}$, le cadriciel doit successivement appeler (i) la fonction d'unmarshalling en passant le type B, (ii) la fonction de conversion de B en A, (iii) la fonction de marshalling en passant le type A pour obtenir un document $\mathtt{doc}_\mathtt{A}$ de type $\mathcal{G}_\mathtt{A}$. Pour le cas SOAP, sur l'entrée $\mathtt{doc}_{\mathcal{F}_\mathtt{B}}$ de type $\mathcal{GF}_\mathtt{B}$, le cadriciel doit successivement appeler (i) la fonction d'unmarshalling en passant le type $\mathcal{F}_\mathtt{B}$, produit par la génération de commande de B, (ii) la fonction de projection, correspondante à l'appel d'un getter de la commande de la classe $\mathcal{F}_\mathtt{B}$, (iii) la fonction de conversion de B en A, (iv) la fonction de plongement en passant le type A, correspondant à l'appel du constructeur de la classe commande de $\mathcal{F}_\mathtt{A}$ et à un setter, (v) la fonction d'unmarshalling en passant le type $\mathcal{F}_\mathtt{A}$ pour obtenir un document $\mathtt{doc}_{\mathcal{F}_\mathtt{A}}$ de type $\mathcal{GF}_\mathtt{A}$.

Algorithme de lifting. Une implémentation directe de la spécification n'est pas satisfaisante. Le cadriciel doit connaître les deux types A et B, ce qui est une contrainte forte. La conversion n'est pas efficace car elle implique une double traduction, à partir des documents vers des objets et dans le sens inverse. Une meilleure solution est de définir directement une transformation de documents en documents. Pour un Data Binder standard, comme JAXB, un document converti suivant un sous-type, $\mathtt{doc}_\mathtt{B}$, diffère d'un document converti suivant un super-type, $\mathtt{doc}_\mathtt{A}$, en deux points :

- le nom de l'étiquette racine, qui réfère normalement au schéma de liaison correspondant au type objet,
- la présence d'éléments supplémentaires en raison de la présence d'attributs supplémentaires définis dans le sous-type.

Pour appliquer cet algorithme, la liaison de schéma doit définir une étiquette unique pour chaque sous-éléments lors du marshalling par récursivité des objets imbriqués, indépendamment des types objets utilisés pour le marshalling. Nous avons besoin de cette condition, afin de réduire la complexité de l'algorithme en réduisant l'écart entre les structures associées à des sous-types et celles associées à leurs super-types. JAXB par exemple, satisfait cette condition. Pour illustrer cette idée, nous considérons l'exemple de la Figure F.23. Dans cet exemple, nous considérons un marshalling d'une instance eMsg de type A' via le type A' à l'émission et un unmarshalling via le type A supertype de A' à la réception. Nous considérons un objet imbriqué, c' dans eMsg, une instance de C', converti suivant le type C' à l'émission et re-converti suivant le type C supertype

de C' à la réception. L'exemple montre comment la balise associée au marshalling de l'objet imbriqué c' ne dépend pas du nom du type de marshalling, C', mais de la liaison de schéma de l'attribut défini dans la classe A, ici c'est x. Ainsi, le document envoyé diffère de celui attendu avec :

- l'étiquette racine (A' à la place de A)
- la présence de deux éléments supplémentaires :
  - k comme un sous-élément de x, résultant du marshalling de l'attribut k de c',
  - i comme un sous-élément de a', résultant du marshalling de l'attribut i de eMsg.

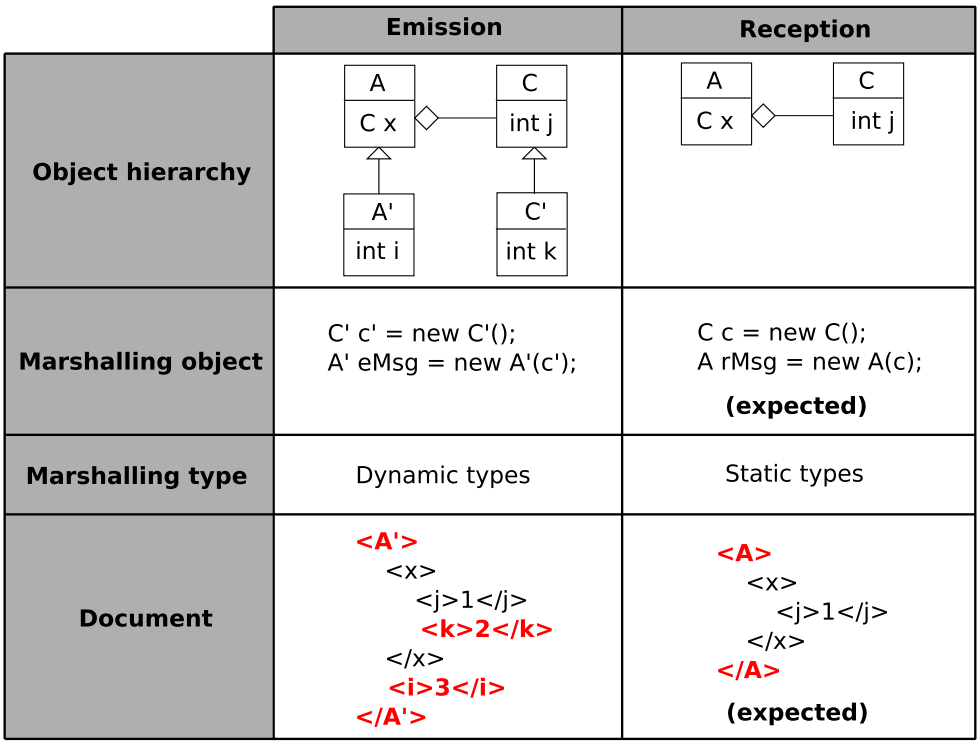


FIG. F.23 – Un exemple de l'interopérabilité avec sous-typage des objets imbriqués

Nous en déduisons les lifting algorithmes suivants :

- **RESTful :** pour transformer un document $\texttt{doc}_\texttt{B}$ en un document $\texttt{doc}_\texttt{A}$ de type $\mathcal{G}_\texttt{A}$, il suffit de renommer si nécessaire l'étiquette racine, et, si possible, d'extraire les sous-documents de $\texttt{doc}_\texttt{B}$ pour correspondre à la définition de $\mathcal{G}_\texttt{A}$. Cette extraction correspond à un algorithme bien connu, le problème d'inclusion d'arbre [13],
- **SOAP :** afin de transformer le document $\texttt{doc}_{\mathcal{F}_\texttt{B}}$ en un document $\texttt{doc}_{\mathcal{F}_\texttt{A}}$ de type $\mathcal{GF}_\texttt{A}$, il suffit d'extraire si possible les sous-documents de $\texttt{doc}_{\mathcal{F}_\texttt{B}}$ pour correspondre à la définition de $\mathcal{GF}_\texttt{A}$. Contrairement au cas RESTful, il n'est pas nécessaire de renommer la balise racine parce que dans SOAP cette balise réfère à l'opération du service. Si la balise racine

ne correspond pas à celle attendue ainsi une erreur doit être détectée parce que le service n'est pas capable d'identifier l'opération appelée.

Déploiement de l'implémentation. Puisque l'algorithme de lifting ne dépend pas du type `B`, connu à l'émission, mais seulement de type `A`, connu à la réception, nous pouvons déterminer le meilleur endroit pour appliquer notre algorithme : c'est à la réception et à l'unmarshalling. Deux implémentations sont possibles pour l'algorithme de lifting :

- **Implémentation 1 :** à l'intérieur de la fonction d'unmarshalling du `Data Binder`,
- **Implémentation 2 :** à l'intérieur du cadriciel orienté objet par un mécanisme d'interception qui applique les adaptations nécessaires sur le document reçu avant de l'envoyer au `Data Binder` pour unmarshalling.

Chaque implémentation présente un avantage comparé à l'autre. Enfin, le choix d'une solution ou de l'autre dépend de chaque contexte et des préférences du développeur. Dans cette thèse, nous avons appliqué le premier choix d'implémentation sur `cxf`.

F.8 Conclusion et perspectives

Pour conclure, cette thèse présente un travail conséquent qui vise à uniformiser et à améliorer le développement des services Web qui mettent en jeu à la fois une couche à services et une couche à objets. Les solutions proposées visent à résoudre les problèmes d'interopérabilité, de couplage des données et du processus de découverte. Pour cela, les contributions de cette thèse sont les suivantes :

- Un modèle formel unifié pour décrire les interactions entre services et le processus de découverte.
- Une API orientée objet pour la découverte dynamique de services conforme aux pratiques du développement orienté objet et un protocole pour unifier le processus de découverte de services en s'appuyant sur des techniques de sous-typage.
- Une spécification reliant les données et les types entre la couche à services et la couche à objets, des schémas de conversion entre ces couches, et une implémentation dans l'environnement `cxf`.

Les perspectives de ce travail concernent notamment des questions liées à l'implémentation et l'expérimentation des propositions, l'extension de la découverte de services pour la prise en compte des protocoles métier des services ou de la sécurité, des corrections à faire sur la génération de schémas lorsque le type générique est utilisé, l'extension des deux technologies `RMI` et `CORBA` avec une couche à services tout en respectant le principe de substitution et finalement l'intégration du type nominal de la couche à objets avec le type structurel de la couche à services défini dans cette thèse notamment pour les règles de sous-typage.

Bibliographie

[1] Web services dynamic discovery (ws-discovery) version 1.1, OASIS standrad. Technical report, July 2009. 14, 32, 248

[2] *RESTEasy JAX-RS - RESTFul Web Services for Java*, chapter 8. JBoss.org Documentation, 2014. 35

[3] Suad Alagic and Philip A. Bernstein. Mapping XSD to OO schemas. In Moira C. Norrie and Michael Grossniklaus, editors, *Proceedings of the 2nd International Conference on Object Databases, ICOODB 2009*, volume 5936 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 2009. 53, 251

[4] Diana Allam et al. Well-Typed Services Cannot Go Wrong. Research Report RR-7899, INRIA, May 2012. 100, 209

[5] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services : Concepts, Architectures and Applications*. Springer, Berlin, 2004. 37

[6] George Athanasopoulos, Aphrodite Tsalgatidou, and Michael Pantazoglou. Interoperability among heterogeneous services. In *Proceedings of the 2006 IEEE International Conference on Services Computing, SCC 2006*, pages 174–181. IEEE Computer Society, 2006. 59, 254

[7] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In Stephen Chong, editor, *Proceedings of the 25th IEEE Computer Security Foundations Symposium, CSF 2012*. IEEE, 2012. 194

[8] T. Bellwood and al. UDDI spec technical committee specification. UDDI Version 3.0, pages 21–22, 87–90. OASIS, July 2004. 14, 30, 74, 120, 248, 289

[9] Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of nosql languages. *ACM SIGPLAN Notices*, 48(1) :101–114, January 2013. 63, 257

[10] Tim Berners-Lee. Universal Resource Identifiers in WWW, A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web (IETF RFC 1630). `http://www.w3.org/Addressing/rfc1630.txt`, June 1994. 28

[11] Gérard Berry and Gérard Boudol. The chemical abstract machine. In Frances E. Allen, editor, *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90*, pages 81–94, New York, NY, USA, 1990. ACM. 60, 255

[12] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1), 1992. 90, 93

[13] Philip Bille and Inge Li Gortz. The Tree Inclusion Problem : In Linear Space and Faster. *ACM Transactions on Algorithms*, 7(3) :38 :1–38 :47, July 2011. 155, 278

[14] Kenneth P. Birman. Like it or not, web services are distributed objects. *Communications of the ACM*, 47(12) :60–62, December 2004. 196

[15] Kenneth P. Birman. *Reliable Distributed Systems : Technologies, Web Services, and Applications*. Springer-Verlag, Secaucus, NJ, USA, 2005. 196

[16] Gérard Boudol. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992. 26, 60, 66, 255

[17] Samuele Carpineti and Cosimo Laneve. A basic contract language for web services. In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 197–213. Springer, 2006. 59, 65, 254

[18] Giuseppe Castagna. Covariance and Contravariance : Conflict without a cause. In ACM, editor, *Transactions on Programming Languages and Systems*, volume 17, pages 1–17, 1995. 73

[19] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the p-calculus. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS '05*, pages 92–101. IEEE Computer Society, 2005. 26, 66

[20] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4) :173–191, 1996. 58

[21] Roberto Chinnici, Marc Hadley, and Rajiv Mordani. The java api for xml-based web services (jax-ws) 2.0. Sun Microsystems, April 2006. 38

[22] William R. Cook and Janel Barfield. Web service versus distributed objects : A case study of performance and interface design. *International Journal of Web Services Research*, 4(3) :49–64, 2007. 52

[23] Microsoft Corporation. Web service security : Scenarios, patterns, and implementation guidance for web services enhancements (wse) 3.0. Patterns and practices, pages 253–269. Microsoft Press, 2006. 62, 290

[24] Wolfgang Emmerich. *Engineering Distributed Objects*. John WILEY, New York, USA, 2000. 12, 246

[25] Wolfgang Emmerich. *Engineering Distributed Objects*, chapter 2, pages 50–58. John WILEY, New York, USA, 2000. 48

[26] Wolfgang Emmerich. *Engineering Distributed Objects*, chapter 3, pages 73–85. John WILEY, New York, USA, 2000. 50, 51, 52, 289

[27] Thomas Erl. *SOA : Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, first edition, 2007. 12, 24, 246

[28] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96*, pages 372–385, New York, NY, USA, 1996. ACM. 60, 67, 255, 257

[29] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping : Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4) :19 :1–19 :64, sep 2008. 26, 65, 67, 97, 99, 100, 192, 257, 260

[30] Aniruddha Gokhale, Bharat Kumar, and Arnaud Sahuguet. Reinventing the wheel ? CORBA vs. web services. In David Lassner, Dave De Roure, and Arun Iyengar, editors, *Proceedings of the 11th International World Wide Web Conference, WWW 2002*, January 2002. 52

[31] Stephen Graham, Glen Daniels, Doug Davis, Yuichi Nakamura, Simeon Simeonov, Toufic Boubez, Ryo Neyama, Peter Brittenham, Paul Freemantle, and Dieter Koenig. *Building Web Services with Java : Making Sense of XML, SOAP, WSDL, and UDDI*, chapter 5. Sams Publishing, 2004. 43

[32] Neil A. B. Gray. Comparison of web services, Java-RMI, and CORBA service implementations. In Jean-Guy Schenider and Jun Han, editors, *Proceedings of the 5th Australasian Workshop on Software and System Architectures at ASWEC 2004*, pages 52–63, April 2004. 52

[33] Martin Gudgin, Marc Hadley, and Tony Rogers. Web services addressing 1.0 - core. W3C, May 2006. 121

[34] Marc Hadley. Web application description language. W3C, August 2009. 33

[35] Dick Hardt. The oauth 2.0 authorization framework. Technical report, Microsoft, 2012. 63, 256

[36] Kejing He. Integration and orchestration of heterogeneous services. In *Proceedings of the 2009 Joint Conferences on Pervasive Computing, JCPC 2009*, pages 467–470. IEEE, 2009. 59, 254

[37] Tom Heath and Christian Bizer. *Linked Data : Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web : Theory and Technology. Morgan & Claypool, first edition, 2011. 14, 34, 248

[38] Glenn Hostetler and Sandor Hasznos. *Web Service and SOA Technologies*, volume 208. Practicing Safe Techs, first edition, 2009. 11, 27

[39] Jadwiga Indulska. Formal methods for distributed processing. chapter Subtyping in distributed systems, pages 233–253. Cambridge University Press, New York, NY, USA, 2001. 52

[40] Nicolai M. Josuttis. *SOA in Practice, The Art of Distributed System Design*, chapter 4, pages 3–34. O'Reilly Media, 2007. 24

[41] Matjaz B. Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, and Ivan Vezocnik. Java rmi, rmi tunneling and web services comparison and performance analysis. *ACM SIGPLAN Notices*, 39(5) :58–65, May 2004. 52

[42] Martin Kalin. *Java Web Services, Up and Running*, chapter 6. O'Reilly, 2009. 43

[43] Kohsuke Kawaguchi, Sekhar Vajjhala, and Joe Fialli. The java architecture for xml binding (jaxb) 2.2. Specification Final release, Sun Microsystems, 2009. 40, 158, 242

[44] Uwe Keller, Holger Lausen, and Michael Stollberg. On the semantics of functional descriptions of web services. In York Sure and John Domingue, editors, *Proceedings of the 3rd European Semantic Web Conference, ESWC 2006*, volume 4011 of *Lecture Notes in Computer Science*, pages 605–619. Springer, 2006. 59, 254

[45] Martin Kempa and Volker Linnemann. Type checking in xobe. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *Proceedings of 10th Conference on Database Systems for Business, Technology and Web, BTW 2003*, volume 26 of *Lecture Notes in Informatics*, pages 227–246. GI, February 2003. 54

[46] Jinhan Kim, Jaejeong Lee, and Byungjeong Lee. Runtime service discovery and reconfiguration using owl-s based semantic web service. In *Proceedings of the 7th International Conference on Computer and Information Technology, CIT 2007*, pages 891–896. IEEE, October 2007. 26

[47] Dimitrios Kourtesis and Iraklis Paraskakis. *Semantic Enterprise Application Integration for Business Processes : Service-Oriented Frameworks*, chapter 4. Business Science Reference, Hersley, 2009. 16, 26, 35, 66, 251

[48] Ralf Lämmel and Erik Meijer. Revealing the x/o impedance mismatch : changing lead into gold. In *Proceedings of the 2006 international conference on Datatype-generic programming, SSDGP'06*, volume 4719 of *Lecture Notes in Computer Science*, pages 285–367, Berlin, Heidelberg, 2007. Springer-Verlag. 42, 53, 244, 251

[49] Leslie Lamport and Nancy Lynch. Distributed computing : models and methods. In *Handbook of Theoretical Computer Science vol B*, pages 1157–1199. MIT Press, 1990. 58, 90, 254, 257

[50] Thomas Y. Lee and David W. Cheung. Formal models and algorithms for XML data interoperability. *Journal of Computing Science and Engineering (JCSE)*, 4(4) :313–349, 2010. 16, 35, 54, 66, 115, 251

[51] Barbara Liskov and Jeannette Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841, 1994. 12, 53, 73, 80, 247

[52] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. In *The Journal of Logic and Algebraic Programming*, volume 70, pages 96–118. Elsevier press, January 2007. 59

[53] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 260–284, Berlin, Heidelberg, 2008. Springer-Verlag. 197

[54] James Mcgovern, Sameer Tyagi, Michael E. Stevens, and Sunil Mathew. *Java Web Services Architecture*, chapter 2, pages 35–61. Elsevier Science (USA), 2003. 24, 25, 26, 289

[55] James Mcgovern, Sameer Tyagi, Michael E. Stevens, and Sunil Mathew. *Java Web Services Architecture*, chapter 1, pages 3–34. Elsevier Science (USA), 2003. 27

[56] Brett McLaughlin. *Java & XML Data Binding*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. 40, 42, 53, 242, 243

[57] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996. 104, 209, 212

[58] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts i and ii. *Information and Computation*, 100 :1–40 & 41–77, 1992. 60, 67, 255

[59] Rajeev Hathi Naveen Balani. Apache cxf web service development- develop and deploy soap and restful web services. Technical report, Birmingham, B27 6PA, UK, 2009. 43

[60] Oasis Consortium. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 11 April, 2007. 37, 63, 256

[61] Kevin R. Page, David C. De Roure, and Kirk Martinez. Rest and linked data : A match made for domain driven development ? In *Proceedings of the 2nd International Workshop on RESTful Design, WS-REST '11*, ACM International Conference Proceeding Series, pages 22–25, New York, NY, USA, 2011. ACM. 14, 34, 248

[62] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, and Sanjay Singh. Formal verification of oauth 2.0 using alloy framework. In *Proceedings of the 2011 International Conference on Communication Systems and Network Technologies, CSNT '11*, pages 655–659, Washington, DC, USA, june 2011. IEEE Computer Society. 194

[63] Joachim Parrow. *An Introduction to the pi-calculus*, pages 479–543. Elsevier, 2001. 60, 255

[64] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services : making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web, WWW '08*, pages 805–814. ACM, 2008. 27

[65] Santiago Pericas-Geertsen and Marek Potociar. Jax-rs : Java api for restful web services, version 2.0 final release. Oracle, May 2013. 38

[66] Ken Q. Pu. Service description and analysis from a type theoretic approach. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 379–386, 2007. 66

[67] Leonard Richardson and Sam Ruby. *RESTful Web Services - Web services for the real world*. O'Reilly Media, 2007. 33

[68] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. *Journal of Automated Reasoning*, 31(3-4) :335–370, 2003. 66, 67

[69] Richard T. Sanders, Rolv Braek, Gregor von Bochmann, and Daniel Amyot. Service discovery and component reuse with semantic interfaces. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *Proceedings of the 12th International SDL Forum, SDL 2005*, pages 85–102, June 2005. 26

[70] Davide Sangiorgi and David Walker. *The Pi-Calculus : A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. 26, 60, 255

[71] Thierry Sans and Iliano Cervesato. Qwesst for type-safe web programming. 3rd International Workshop on Logics, Agents, and Mobility (LAM'10) Edinburgh, Scotland, http ://www.qatar.cmu.edu/ tsans/index.php ?page=research, 2010. 59, 66, 67, 254

[72] João Costa Seco and Luís Caires. A basic model of typed components. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming, ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, pages 108–128. Springer Berlin Heidelberg, 2000. 35, 65, 101

[73] Software Security. Four Attacks on OAuth - How to Secure Your OAuth Implementation. http://software-security.sans.org/blog/2011/03/07/oauth-authorization-attacks-secure-implementation, 2011. 194

[74] Fredrik Seehusen and Ketil Stølen. Information flow security, abstraction and composition. *Information Security, IET*, 3(1) :9–33, 2009. 59, 254

[75] Peter Sewell. Global/local subtyping and capability inference for a distributed pi-calculus. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium, ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 695–706, 1998. 26, 61

[76] Youngmee Shin and Hyunjoo Bae. Web service providing using web service transformation. *World Academy of Science, Engineering and Technology*, pages 890–896, 2010. 59, 254

[77] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindstrom. Json-ld 1.0 - a json-based serialization for linked data. W3C, Septembre 2013. 35

[78] Satish Narayana Srirama, Matthias Jarke, and Wolfgang Prinz. Mobile web service provisioning. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services, AICT/ICIW 2006*, page 120, Los Alamitos, CA, USA, 2006. IEEE Computer Society. 26, 248

[79] Sun. Java management extensions instrumentation and agent specification, v1.2. Specification v1.2, Sun Microsystems, 2002. 171

[80] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus : a model of service oriented computation. In Sophia Drossopoulou, editor, *Proceedings of the 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*. Springer, 2008. 59

[81] Steve Vinoski. CORBA : Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Communications Magazine, IEEE*, 35(2) :46–55, 1997. 50, 196

[82] Werner Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6) :59–66, November 2003. 52, 196

[83] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *Computing Systems*, 9(4) :265–290, 1996. 50, 196

[84] Sonja Zaplata, Viktor Dreiling, and Winfried Lamersdorf. Realizing mobile web services for dynamic applications. In Claude Godart, Norbert Gronau, Sushil Sharma, and Gerome

Canals, editors, *Proceedings of the 9th IFIP Conference on e-Business, e-Services, and e-Society, I3E 2009*, volume 305 of *IFIP Advances in Information and Communication Technology*, pages 240–254. Springer, 9 2009. 26, 248

[85] A. Zisman, G. Spanoudakis, and J. Dooley. A framework for dynamic service discovery. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08*, pages 158–167, Washington, DC, USA, 2008. IEEE Computer Society. 26, 248

Table des figures

Liste des tableaux

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes

Thèse de Doctorat

Diana ALLAM

Couplage Faible et Principe de Substitution dans les Environnements à Objets pour les Services Web

Loose Coupling and Substitution Principle in Object-Oriented Frameworks for Web Services

Résumé

Actuellement, l'implémentation des services (modèles `SOAP` et `RESTful`) et de leurs applications clientes est de plus en plus basée sur la programmation par objet. Ainsi, les cadriciels orientés-objets pour les services Web sont essentiellement composés de deux couches : une couche à objets qui enveloppe une couche à services. Dans ce contexte, deux principes sont nécessaires pour la spécification de ces cadriciels : (i) En premier lieu, un couplage faible entre les deux couches, ce qui permet de cacher la complexité des détails techniques de la couche à services dans la couche à objets et de faire évoluer la couche à services avec un impact minimal sur la couche à objets (ii) En second lieu, une interopérabilité induite par le principe de substitution associée au sous-typage dans la couche à objets. Dans cette thèse, nous présentons d'abord les faiblesses existantes dans les cadriciels orientés-objets liés à ces deux principes. Ensuite, nous proposons une nouvelle spécification pour ces cadriciels en vue de résoudre ces problèmes. Comme application, nous décrivons la mise en oeuvre de notre spécification dans le cadriciel `cxf`, à la fois pour `SOAP` et `RESTful`.

Mots clés

Architecture Orientée-Services, Programmation par Objet, Interopérabilité, Couplage Faible, Sous-typage, Modèle de Passation de Messages

Abstract

Today, the implementation of services (`SOAP` and `RESTful` models) and of client applications is increasingly based on object-oriented programming languages. Thus, object-oriented frameworks for Web services are essentially composed with two levels: an object level built over a service level. In this context, two properties could be particularly required in the specification of these frameworks: (i) First a loose coupling between the two levels, which allows the complex technical details of the service level to be hidden at the object level and the service level to be evolved with a minimal impact on the object level, (ii) Second, an interoperability induced by the substitution principle associated to subtyping in the object level, which allows to freely convert a value of a subtype into a supertype. In this thesis, first we present the existing weaknesses of object-oriented frameworks related to these two requirements. Then, we propose a new specification for object-oriented Web service frameworks in order to resolve these problems. As an application, we provide an implementation of our specification in the `cxf` framework, for both `SOAP` and `RESTful` models.

Key Words

Service-Oriented Architecture, Object-Oriented Programming, Interoperability, Loose Coupling, Subtyping, Message-Passing Model