UNIVERSITÉ DU QUÉBEC À MONTRÉAL

Composition de services web par appariement de signatures

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ANISS ALKAMARI

Janvier 2008

Remerciements

Cet espace ne peut me suffire à remercier tous les gens qui le méritent. Que les personnes qui n'y sont pas citées m'en excusent et acceptent mes remerciements les plus sincères.

Je commencerai par remercier monsieur Hafedh Mili, professeur à l'Université du Québec à Montréal, qui m'a permis de faire partie du laboratoire LATECE (LAboratoire de recherche sur les TEchnologies du Commerce Électronique) et a supervisé cette recherche. Sa disponibilité, sa persévérance, son perfectionnisme ainsi que ses encouragements m'ont beaucoup aidé tout au long de ce travail. Je garderai spécialement un doux souvenir de son approche amicale et de son humour peu commun. Que ce mémoire lui soit le modeste témoignage de ma reconnaissance et de mon plus grand respect.

Je remercie aussi tous mes collègues du groupe du LATECE, en particulier Ghizlane, Mohand, Mélanie, Romdhane, Redouane, Sébastien et Christian, qui ont répondu à mes questions, supporté mes petites angoisses, mes remises en question et accepté mes petites excentricités.

Je tiens finalement à exprimer ma profonde gratitude à mon père Bachir, ma mère Noufissa, mes frères Nawfal et Karim, ainsi que ma sœur Schada, qui m'ont toujours soutenu dans les moments les plus difficiles et sans qui je n'y serais jamais arrivé.

Merci à tous. Ce fut un long chemin.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

Les services web ont longtemps été présentés comme la réponse, tant attendue, à l'interopérabilité souhaitée des systèmes distribués hétérogènes. Dans le passé, plusieurs technologies ont fait la promesse d'offrir cette interopérabilité : .NET, DCOM, J2EE, CORBA, etc. La promesse ne fut jamais tenue, tantôt parce que la technologie en question n'était pas *extensible (adaptable à différentes échelles)* (DCOM et CORBA), tantôt parce qu'elle était de propriété industrielle (DCOM, .NET, etc.). UDDI *(Universal Discovery Description and Integration)* publie tous les services web disponibles et facilite ainsi la requête des services offerts par les différentes entreprises. Néanmoins, la façon dont ces requêtes sont formulées laisse à désirer. En particulier, UDDI prend pour acquis que, pour chaque besoin commercial, il y aurait un service commercial. Cette réalité a rapidement convaincu les utilisateurs des services web de l'importance d'en faire la composition. Par conséquent, la composition des services web a connu beaucoup d'intérêt dans les dernières années. Les approches adoptées pour composer des services web sont différentes. Nous prônons une approche syntaxique. Nous pensons qu'une recherche par contexte et domaine d'industrie permettrait une découverte adéquate de services web satisfaisant les besoins du client. WSDL nous facilite la tâche, puisque les types de ses éléments sont des documents d'affaires donnant une bonne idée des services qu'ils offrent. Nous utilisons l'appariement des signatures des opérations pour chercher l'ensemble d'opérations fournissant les types dont on a besoin. La composition des services web devient une composition de fonctions qui, partant d'un ensemble de messages d'entrées, produisent un ensemble de messages de sorties. Dans cette recherche, nous présentons un algorithme qui se base sur différentes manières d'apparier les types et qui satisfait cette approche sémantique ainsi que les résultats trouvés.

MOTS-CLÉS: Services Web, .NET, DCOM, J2EE, CORBA, standard UDDI, WSDL, appariement, composition.

CHAPITRE 1

INTRODUCTION

À son tout début, le commerce électronique était essentiellement fondé sur la vente de services et de biens aux clients. On était à l'ère du B2C (*Business to Customer*). Toutefois, on se rendit rapidement compte que l'échange de services entre entreprises elles-mêmes est, non seulement nécessaire, mais surtout souhaitable et lucratif. Ce fut la naissance du B2B (*Business to Business*). Selon le *Computer Industry Almanac*, en 2002, la population mondiale des internautes était de 490 millions de personnes. Alors qu'en mars 2006, le nombre d'internautes dans le monde, âgés de 15 ans et plus, serait passé à 694 millions de personnes, selon les chiffres publiés par *ComScore*. Aujourd'hui, même si toutes les entreprises n'ont pas fondé l'essentiel de leurs activités économiques sur le net, elles se doivent au moins d'y être représentées, ne serait-ce que pour donner l'image d'une entreprise moderne, dynamique et technologiquement à la page (Nicolle, 2002). Internet leur facilite grandement les choses et permet aux entreprises d'échanger des données à distance et de les interroger. En particulier, les services web permettent une communication facile entre ces entreprises et leurs partenaires et clients. Grâce à la norme UDDI, un service web est enregistré dans un registre puis invoqué par un utilisateur (*consumer*). Pour combler certains besoins du client, il est impératif de composer des services web en un flux de services web satisfaisant le besoin recherché. Le but de notre travail est de proposer une technique de

composition des services web fondée sur l'appariement des types, sans aucune manipulation des normes utilisées par les services web.

1.1. Découverte des services web

Commençons par illustrer la problématique. Soit A une agence de voyages. A offre un service de réservation dite groupée ou globale, réunissant un billet d'avion, une location de voiture et une chambre d'hôtel. Un prospère chef d'entreprise de Montréal est justement à la recherche d'un voyage à Bali dans une suite d'un hôtel cinq étoiles, assorti d'une location de voiture au meilleur prix possible. Il contacte un agent de l'agence A et demande son aide. L'agent effectue ses recherches, vérifie que les dates concordent avec les exigences du client, compare les prix et communique les résultats au client.

Pour que ces recherches soient possibles, le système que l'agent utilise doit effectuer les tâches suivantes :

- Faire la découverte des services web qui offrent les disponibilités des billets Montréal-Denpasar aller-retour ainsi que leur prix et qui permettent leur réservation.
- Choisir le service web qui offre le billet le moins cher.
- Découvrir les services web qui offrent des chambres d'hôtel de cinq étoiles à Bali et permettent leur réservation.
- Choisir le service web qui offre le meilleur prix de chambres cinq étoiles à Bali.
- Découvrir les services web qui offrent la location de voitures à Bali et permettent leur réservation.
- Choisir le service web qui offre la location de voitures à Bali au prix le moins cher.
- Composer les services web choisis en un flux d'activités permettant d'exécuter toutes ces réservations tout en prévoyant le cas d'échec.
- Exécuter le processus de réservations.

Ce scénario est actuellement impossible à réaliser. On voudrait que cet enchaînement de tâches se fasse dynamiquement, car les besoins du client ne sont pas figés et connus d'avance. Mais tels que décrits en ce moment, les services web ne permettent pas l'interaction entre services, leur découverte dynamique ou automatique, ou encore leur négociation et leur composition pour former d'autres services sans une intervention humaine.

En effet, aucune norme utilisée par les services web ne permet cette composition automatique, puisqu'aucune n'offre une description suffisante des services. UDDI permet certes la découverte des services web, mais ne permet pas le choix du meilleur fournisseur d'un service web parmi plusieurs fournisseurs d'un même service. UDDI ne permet pas non plus la composition dynamique d'un ensemble de services web fondée sur un lien logique. Srivastava et Koehler (2003) indiquent que les règles d'affaires avec les partenaires et la définition de leurs interactions sont encore statiques. Plusieurs approches de composition des services web ont été explorées. Nous les présentons brièvement.

1.2. Approches existantes

Il existe principalement deux grandes approches de composition des services web : l'approche statique et l'approche dynamique. Nous présentons brièvement chacune de ces approches dans ce qui suit.

1.2.1. Composition statique des services web

Dans cette approche, les services web à composer sont choisis *à l'heure de faire l'architecture et le design. Les composants sont choisis et reliés ensemble, avant d'être compilés et déployés* (Dustdar et Schreiner, 2005). Ceci peut marcher en autant que l'environnement des services web, les partenaires d'affaires, ainsi que les dits composants changent peu ou pas du tout. *Microsoft Biztalk* et *Bea Weblogic* sont deux exemples de *moteurs* de composition statique de services web (Sun et al., 2003). Si les fournisseurs de services proposent d'autres services ou changent les anciens services, des incohérences peuvent être causées, ce qui demanderait un changement de l'architecture du logiciel, voire de la définition du processus et créerait l'obligation de faire une nouvelle conception du système. Dans ce cas, la composition statique des services web est considérée trop restrictive : les composants doivent s'adapter automatiquement aux changements imprévisibles (Sun et al., 2003).

1.2.2. Composition dynamique des services web

Dans cette approche, les services web à composer sont choisis au moment de l'exécution, durant laquelle une recherche des services est effectuée dans les registres Channa

et al. À titre d'exemple, Sun et al. (2003) présentent une plateforme de composition dynamique des services web : StarWSCOP (*Star Web Services Composition Platform*). Pour faire de la composition dynamique des services web, StarWSCOP effectue les quatre étapes suivantes (Dustdar et Schreiner, 2005) :

- Les fournisseurs des services web publient leurs services dans un registre.
- StarWSCOP décompose les requêtes des utilisateurs en services abstraits et envoie une requête SOAP au registre pour trouver les services appropriés.
- Le registre de services web disponibles renvoie une liste de services web concrets.
- StarWSCOP envoie une requête SOAP aux services web trouvés, ensuite se lie avec eux.

StarWSCOP contient plusieurs modules (Dustdar et Schreiner, 2005) : un système dit intelligent pour décomposer les requêtes des utilisateurs en plusieurs services abstraits, un moteur de recherche de services web pour découvrir les services web qui respectent les conditions de l'utilisateur, un moteur de composition qui exécute les services en ordre, un estimateur QoS (*Quality of Service*) qui mesure en temps réel les métriques de QoS du service composé, etc. Pour supporter les métriques QoS de cette composition dynamique de services web, on rajoute au WSDL des attributs de QoS (temps, coût, fiabilité, etc.). La Figure 1.1 nous en donne un aperçu :

```
<portType name = "GetTotalCost">
        <operation name = "getTotalCost"
                Cost = "25"
                Time = "60"
                Reliability = "85%">
        …
        </operation>
</portType>
```

Figure 1.1. Exemple d'attributs Qos rajoutés à la description d'un WSDL

Une couche axée ontologie est aussi rajoutée à UDDI afin de faire de l'appariement sémantique pour les services web (Dustdar et Schreiner).

1.3. Notre approche

Le but de notre recherche est de proposer un mécanisme de composition de services web par appariement de types. La composition que nous proposons est dynamique. Elle ne demande aucun rajout aux fichiers WSDL et aucune manipulation sur le registre UDDI. À partir d'un ensemble de départ composé d'un ou plusieurs types, nous voulons atteindre un ensemble d'arrivée composé d'autres types. Pour y arriver, nous composerons un flux de services web découverts au fur et à mesure grâce aux types disponibles à chaque étape.

Sur le plan pratique, nous mettons en œuvre l'algorithme *EnumerateRealizations* présenté dans « *Intelligent Component Retrieval for Software Reuse* » (Mili et al.) et le testons sur différents fichiers WSDL. Nous utilisons l'analyseur syntaxique (*parser*) WSDL4J pour lire et analyser ces fichiers WSDL.

1.4. Plan du mémoire

Dans le *deuxième* chapitre, nous parlons d'applications distantes, d'approches B2B et B2C, de systèmes distribués avant de présenter les services web et les problèmes existants dans ce domaine. Ensuite, nous présentons la problématique à laquelle nous nous intéressons.

Dans le *troisième* chapitre, nous présentons les standards utilisés par les services web : SOAP, UDDI et WSDL.

Dans le *quatrième* chapitre, nous discutons des approches déjà proposées pour faire de la composition de services web.

Dans le *cinquième* chapitre, nous expliquons notre approche de compositions de services web. Nous présentons notre hypothèse de base et les algorithmes de notre programme.

Dans le *sixième* chapitre, nous expliquons comment nous avons mis en œuvre notre solution, nous présentons les bibliothèques utilisées avant d'exposer les résultats de nos tests.

Dans le *septième* chapitre, nous présentons les différents algorithmes utilisés pour l'appariement des signatures des fonctions et nous expliquons leurs fondements.

Enfin, nous terminons ce mémoire par une conclusion générale où nous faisons une brève synthèse des approches utilisées jusqu'à présent, et nous présentons les limites de notre approche et les suites qu'il serait possible de donner à notre travail.

CHAPITRE 2

TECHNOLOGIES ET STANDARDS UTILISES POUR LES SERVICES WEB

2.1. Besoins : applications distantes utilisées, technologies B2B et B2C

À son tout début, le commerce électronique était essentiellement fondé sur la vente de services et de biens aux clients. On était à l'ère du B2C (*Business to Customer*). Toutefois, on se rendit rapidement compte que l'échange de services entre entreprises elles-mêmes est, non seulement nécessaire, mais surtout souhaitable et lucratif. Ce fut la naissance du B2B (*Business to Business*). Selon le *Computer Industry Almanac*, en 2002, la population mondiale des internautes était de 490 millions de personnes. Alors qu'en mars 2006, le nombre d'internautes dans le monde, âgés de 15 ans et plus, serait passé à 694 millions de personnes, selon les chiffres publiés par *ComScore*. Aujourd'hui, même si toutes les entreprises n'ont pas basé l'essentiel de leurs activités économiques sur le net, elles se doivent, au moins, d'y être représentées, ne serait-ce que pour donner l'image d'une entreprise moderne, dynamique et technologiquement à la page (Nicolle, 2002).

Cette nouvelle réalité a obligé les entreprises à conceptualiser de nouveau leurs processus d'affaires ainsi que leurs technologies d'information, afin d'atteindre une plus grande efficacité dans le monde du commerce électronique. Kalatoka et Robinson (1999) parlent de cette *reconception* comme « *une complexe fusion de processus d'affaires, des*

applications des entreprises et une restructuration organisationnelle, pour atteindre un modèle d'affaire de haut niveau capable de capitaliser sur les opportunités offertes par Internet comme médium d'activité commerciale. »

Internet offre plusieurs formes de commerce (Grigoryan, 2006) :

- *Business-to-Business* e-commerce (B2B) qui se réfère aux activités commerciales entre deux ou plusieurs institutions.

- *Business-to-Customer* e-commerce (B2C) qui se réfère aux activités commerciales entre les institutions financières et leurs clients (exemple : Yahoo.com, Amazon.com, etc.).

- *Customer-to-Customer* e-commerce (C2C) qui se réfère aux activités commerciales entre plusieurs clients à travers des sites — parties-tiers — (ex : ebay.com).

- *Customer-to-Business* (C2B) qui se réfère, quant à lui, à un regroupement de clients intéressés par l'achat d'une entreprise. Les clients ont, dans ce cas, ou bien un intérêt économique commun (ex : demande d'achat de Mercata.com) ou encore un intérêt commun social (ex : le groupe d'avocat intéressé par Voxcap.com) (Rayport et Jaworski, 2001).

Dans ce qui suit, nous nous intéressons brièvement aux approches B2C et B2B.

2.1.1. L'approche B2C

B2C est *la vente en ligne à différents clients qui peuvent transiger par le site web de la compagnie* (He et al., 2003, p. 986). Internet a joué un grand rôle dans le déploiement du commerce B2C durant les années 1990. En effet, étant désormais un outil « démocratisé » dans la plupart des pays industrialisés et disponible dans beaucoup de foyers, Internet a permis la consultation des sites web, la comparaison des prix et l'achat des produits souhaités par les consommateurs. Cependant, cette activité a connu un certain ralentissement peu de temps après l'année 2000. Plusieurs facteurs en sont responsables : le non-respect des délais

promis et des principes du marketing (Agarwal et al., 2001), le financement inadéquat (Cummings et Carr, 2001), ou encore le mauvais déploiement de certaines stratégies de commerce (Kemmler et al., 2001).

B2C offre certains avantages non négligeables. Par exemple, le magasinage peut être plus rapide (pas de déplacement physique) et plus pratique. Il est aussi plus facile de comparer les prix.

2.1.2. L'approche B2B

Selon *The European Information Technology Observatory* (EITO), B2B est « *l'utilisation de la technologie d'Internet pour mener à terme des relations ou des transactions commerciales à l'interne, avec des fournisseurs, ou avec d'autres entreprises* ». Plus simplement, nous définissons B2B comme *les échanges électroniques entre deux ou plusieurs entreprises, incluant notamment les relations commerciales interentreprises (Frayret, J-M., 2002)*. Les compagnies œuvrant dans le B2B peuvent être catégorisées en plusieurs types :

- Des compagnies offrant leurs services exclusivement via Internet. Telles compagnies exigent l'inscription de leurs clients et opèrent via la poste pour la livraison de la marchandise, ou via le net pour offrir d'autres services (conseils financiers, immobiliers, etc.).

- Compagnies de courtage qui agissent en tant qu'intermédiaires entre une personne voulant un certain service et le fournisseur du service.

- Compagnies d'information qui fournissent une information sur une industrie spécifique pour les compagnies intéressées et les employés de ces dernières.

Quelles sont les différences entre les activités B2B et B2C ? Nous faisons une brève comparaison entre les deux dans ce qui suit.

2.1.3. Brève comparaison entre B2B et B2C

B2B et B2C ne s'adressent pas à la même clientèle. Par conséquent, leur approche marketing, les outils pour atteindre leurs objectifs de vente, la relation entre l'acheteur et le vendeur sont différents. Nous énumérons ici quelques différences importantes :

- Dans l'approche B2C, c'est le consommateur qui visite le site de la compagnie. Dans B2B, ce sont les fournisseurs qui visitent les potentiels acheteurs pour leur offrir leurs services et produits.

- Dans B2C, les acheteurs achètent dans leur temps libre ce dont ils ont besoin. Dans B2B, les acheteurs sont des professionnels payés pour faire le meilleur achat.

- Dans B2C, les acheteurs font, en général, des petits achats pour minimiser les risques. Dans B2B, tous les achats sont d'un prix relativement élevés et un seul mauvais achat pourrait coûter le poste de l'acheteur.

Quelques études révèlent qu'à la fin de l'année 2005, le marché de l'e-commerce en Europe par exemple, (incluant B2B et B2C), valait plus que mille milliards d'Euros (dans l'Europe des 15, plus la Norvège et la Suisse), B2B ayant la plus grande part du gâteau, avec presque mille milliards (Gerhards, 2006). On estime que le marché doublerait en 2009 et que B2C aurait une plus grande expansion. Il n'aura cependant que 20 % de la part du marché.

Mais, en quoi les services web sont-ils liés au e-commerce et plus précisément aux approches B2B et B2C ? C'est ce dont nous parlons dans le prochain paragraphe.

2.1.4. Les services web et les activités B2B et B2C

Nous parlons plus en détail des services web dans les chapitres qui suivent, mais il est pertinent ici de faire le lien entre les deux activités présentées et les services web.

D'abord, il est intéressant de faire le constat suivant : au début, les transactions sur le web étaient bâties en se basant essentiellement sur une interprétation et une utilisation

humaines. Une automatisation des transactions est devenue nécessaire devant l'émergence grandissante de B2B. Les services web sont primordiaux pour cette automatisation.

De plus, grâce aux services web, il est devenu possible d'utiliser les systèmes d'affaires par tout le monde, partout dans le monde, à n'importe quelle heure et sur n'importe quel type de plate-forme. Ces possibilités qu'ouvrent les services web ne sont pas des moindres. De nos jours, les fournisseurs peuvent offrir leurs services déployés sous forme de services web, en publiant leurs spécifications dans un registre accessible aux clients potentiels. Ainsi, ces clients (ou consommateurs) pourront eux-mêmes faire la découverte des services web, choisir les produits et services qui les intéressent et les combiner à leur guise pour atteindre l'objectif ou le service qu'ils désirent s'offrir (Friesen et Namiri, 2006). Par conséquent, les services web facilitent l'ouverture des marchés des compagnies œuvrant dans les B2B et B2C et font en sorte que le commerce électronique soit plus accessible tant aux utilisateurs non spécialisés (B2C) qu'aux autres utilisateurs (B2B).

2.2. *Les technologies distribuées*

De 1945 (début de l'ère moderne des ordinateurs) jusqu'en 1985, le monde du commerce électronique n'a connu que les technologies centralisées assez coûteuses en temps et en argent. C'est au début des années 1980 que les micro-ordinateurs apparurent, faisant ainsi démocratiser l'acquisition des ordinateurs puisque les micro-processeurs étaient désormais produits en quantité, ce qui faisait baisser les coûts. L'avènement des réseaux locaux (LAN) permit l'interconnexion d'un grand nombre de machines et le transfert d'informations à 10 Mbits par seconde. Par conséquent, il devint plus facile de faire interagir des ordinateurs entre eux : le système distribué est né. Mais qu'est-ce qu'un système distribué ?

2.2.1. Définition

Plusieurs agents logiciels combinent leur travail pour faire un système distribué. Pour ce faire, ils communiquent à travers des réseaux contenant matériels et logiciels informatiques. Ce genre de communication fait en sorte que les systèmes distribués offrent une communication plus lente et parfois moins fiable que celle offerte par les systèmes

utilisant une mémoire partagée. Les conséquences de ce manque de rapidité ou de fiabilité sont non négligeables, puisque les développeurs des infrastructures et des applications se doivent de prévoir certaines lacunes imprévisibles liées à l'accès à distance et d'autres questions traitant de la simultanéité ou de la possibilité d'échec partiel (W3C).

Verissimo et Rodrigues (2001) définissent les systèmes distribués comme « un système composé de plusieurs ordinateurs qui communiquent via un réseau, hébergeant des processus qui, à leur tour, utilisent un ensemble de protocoles facilitant l'exécution cohérente de plusieurs tâches/activités distribuées ».

Les systèmes distribués ont présenté un certain attrait dès leur découverte. Pourquoi ? Qu'offrent-ils comme avantages par rapport aux systèmes centralisés ? Ont-ils des inconvénients ? C'est ce dont nous parlerons dans la prochaine section.

2.2.2. Avantages et inconvénients des systèmes distribués par rapport aux systèmes centralisés

Avantages des systèmes distribués

Nous énumérons ci-dessous quelques uns des avantages les plus intéressants des systèmes distribués (Morin, 2001):

- Les systèmes distribués sont plus économiques à utiliser que les systèmes centralisés. Le prix des microprocesseurs, ainsi que leur haute performance, y sont pour beaucoup.

- Les systèmes distribués possèdent une puissance de calcul plus élevée que celles des systèmes centralisés.

- La distribution des systèmes peut accompagner une distribution naturelle de certains domaines : le domaine bancaire, par exemple, et ses succursales éparpillées géographiquement, ou encore, les systèmes de contrôle de chaîne de production.

- La disponibilité des systèmes distribués est beaucoup plus élevée que celle des systèmes centralisés : si une machine tombe en panne, les autres machines peuvent toujours prendre le relai.

- Le partage des données est grandement encouragé par les systèmes distribués. Certaines applications ne peuvent se passer de ce partage de données. (Exemples : Réservations de billets d'avion, inscription aux cours universitaires, etc.).

- Les systèmes distribués permettent le partage de périphériques coûteux, notamment les périphériques d'archivage ou les imprimantes laser.

- Les systèmes distribués encouragent la communication asynchrone et la modification des documents échangés : concepts très importants pour l'échange de courriers électroniques, par exemple.

Inconvénients des systèmes distribués

Dans une définition ludique, Leslie Lamport (créateur de LaTex, entre autres - voir (Lamport)) dit qu'« *un système distribué est un système qui vous empêche de travailler quand une machine, dont vous n'avez jamais entendu parler, tombe en panne* » (Verissimo et Rodrigues, 2001). Nous présentons ici quelques inconvénients des systèmes distribués (Morin, 2002) :

- Le concept de systèmes distribués étant relativement nouveau, les concepteurs de logiciels pour systèmes distribués ont peu d'expérience dans leur mise en œuvre, leur utilisation et leur maintenance.

- En ce qui concerne la communication asynchrone dont on a parlé dans les avantages, il arrive qu'on perde certains messages. Le réseau de communication, quant à lui, est de plus en plus saturé.

- La sécurité est un des problèmes majeurs dans les systèmes distribués : plus le partage de données est facile, plus il faut se soucier de la confidentialité de certaines données jugées hautement importantes et d'ordre privé.

2.3. Les services web

2.3.1. Introduction

Un service web est *une interface qui décrit une collection d'opérations accessibles sur le web via des messages XML standardisés* (Gottschalk et al., 2002). Le service web effectue une ou plusieurs tâches et est décrit selon un standard nommé *service description,* qui fournit tous les détails nécessaires pour interagir avec le service comme les formats du message, les protocoles de transport, la localisation du service, etc.

Les services web s'appuient grandement sur les systèmes distribués. Dans un environnement ouvert, fondé sur la collaboration et la communication entre diverses applications, les services web sont la plate-forme pour l'intégration d'applications. Ces applications sont bâties autour de plusieurs services web fonctionnant ensemble et provenant de sources différentes. Comme le montre la Figure 2.1, l'architecture des services web est une architecture axée-service qui comprend trois composantes et trois opérations. Le fonctionnement d'un service web se passe ainsi (Gottschalk et al., 2002).

- Le fournisseur de services (*Service Provider*) crée le service web et sa définition, puis le publie à l'aide d'UDDI (*Universal Discovery Description and Integration*).

- L'utilisateur des services (*Service Consumer*) peut alors le découvrir via l'interface d'UDDI en utilisant SOAP.

- Le répertoire d'UDDI fournit à l'utilisateur une description WSDL du service et une URL (*Uniforme Resource Locator*) qui mène au service lui-même. C'est cette information que l'utilisateur pourra utiliser pour se connecter au service et l'invoquer.

La suivante Figure 2.1illustre bien le fonctionnement d'un service web.

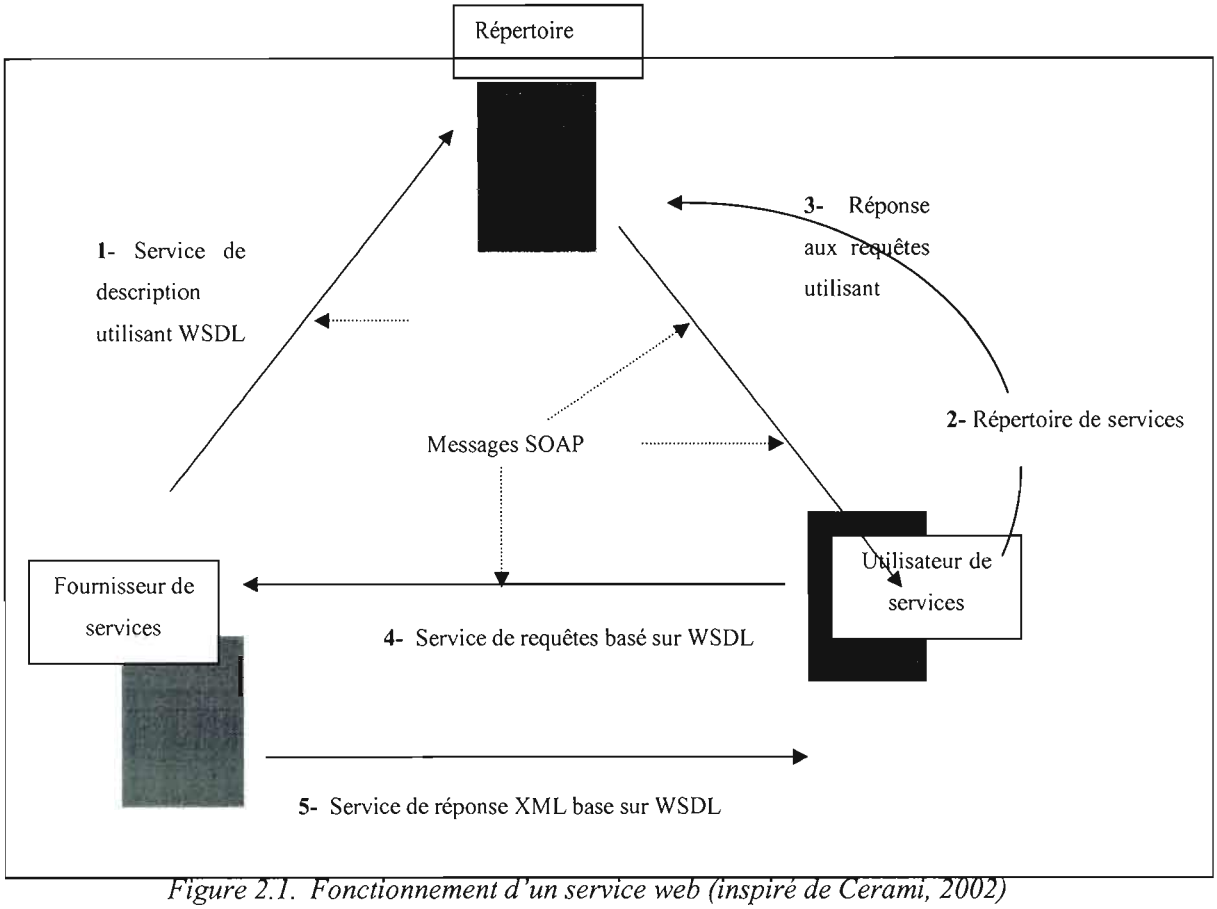


Figure 2.1. Fonctionnement d'un service web (inspiré de Cerami, 2002)

Dans cette figure, nous présentons les normes utilisées et liées aux services web.

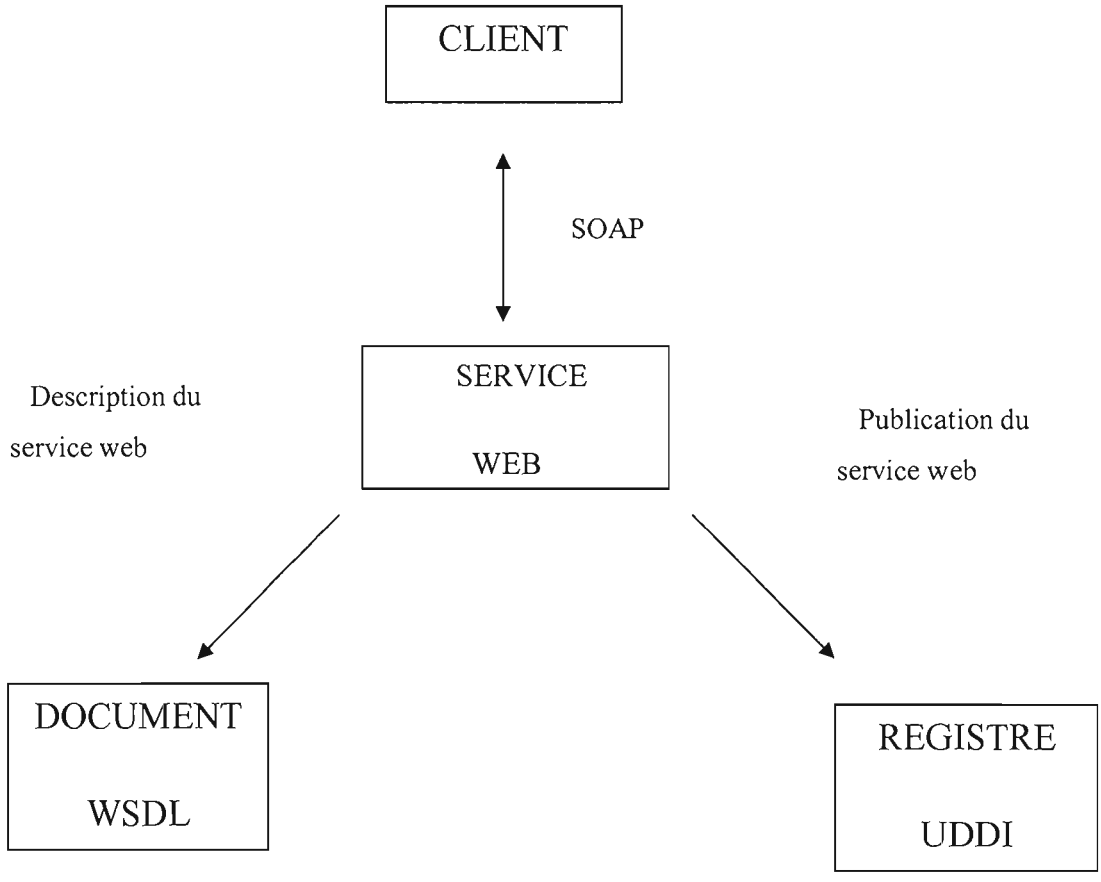


Figure 2.2. Protocoles liés aux services web

2.3.1.1 Avantages des services web

De nos jours, la technologie des services web est populaire et couramment utilisée car elle offre des avantages intéressants pour les utilisateurs des systèmes distribués :

- Les services web réduisent le temps de mise en marché des services offerts par les diverses entreprises.
- Les services web permettent à des programmes écrits en des langages différents et sur des plates-formes différentes de communiquer entre eux par le

biais de certaines normes (Wolter, 2001). En d'autres termes, les services web permettent une meilleure interopérabilité entre les logiciels.

- Les services web utilisent des normes et protocoles ouverts.
- Grâce au protocole HTTP, les services web peuvent fonctionner malgré les pare-feu sans pour autant nécessiter des changements sur les critères de filtrage.
- Les protocoles et les formats de données sont offerts, le plus possible, en format texte pour que la compréhension du fonctionnement des échanges soit plus intuitive.
- Grâce aux services web, les coûts sont réduits par l'automatisation interne et externe des processus commerciaux (Nicolle, 2001).

2.3.1.2 Inconvénients des services web

La technologie des services web comporte plusieurs inconvénients dont :

- Problèmes de sécurité : Il est facile de contourner les mesures de sécurité mises en place par les pare-feu — l'utilisation du protocole *HTTP* (tel que mentionné ci-haut) n'a pas que des avantages — car les normes de sécurité des services web laissent encore à désirer. CORBA, par exemple, qui est une technologie plus mûre, est plus sécuritaire.
- Problèmes de performance : Les services web sont encore relativement faibles par rapport à d'autres approches de l'informatique répartie telles que CORBA ou RMI.
- Technologies connexes : SOAP, technologie primordiale dans le domaine des services web, manque de mécanismes d'authentification, d'autorisation et de chiffrage.

- Confiance : Les relations de confiance entre différentes composantes d'un service web sont difficiles à bâtir, puisque parfois ces mêmes composantes ne se connaissent même pas.

- Syntaxe et sémantique : On se concentre beaucoup sur *comment* invoquer des services (*syntaxe*) et pas assez sur ce que les services web offrent (*sémantique*).

- Fiabilité : Il est difficile de s'assurer de la fiabilité d'un service car on ne peut garantir que ses fournisseurs ainsi que les personnes qui l'invoquent travaillent d'une façon fiable.

- Disponibilité : Les services web peuvent bien satisfaire un ou plusieurs besoins du client. Seront-ils pour autant toujours disponibles et utilisables ? Ça reste un défi pour les services web.

2.3.2. Problèmes existants dans le domaine des services web

2.3.2.1 Problèmes connexes à la qualité de service (QoS)

Découvrir un service web qui nous intéresse est une chose, découvrir le service web le plus adéquat en est une autre. La qualité de service dans le cas des services web se mesure à l'aide de plusieurs métriques dont les métriques de performance, de disponibilité et de fiabilité. Prenons l'exemple du voyageur qui désire acheter un billet d'avion, louer une voiture et une chambre d'hôtel. Une recherche sur UDDI lui permet certainement de trouver plusieurs services web qui remplissent ses critères. Mais lequel sera le plus adéquat, le moins cher, le plus luxueux, le plus rapide, etc., bref, le *meilleur* ? Lequel de ces services sera le plus disponible, le plus fiable ? Lequel aura un temps réponse *acceptable* ? Il devient ainsi nécessaire de choisir les services web pertinents parmi ceux trouvés et de se fixer des critères pour choisir les meilleurs.

2.3.2.2 Problèmes de sémantique

Tel que décrits en ce moment, les services web ne permettent en aucun cas l'interaction entre services, leur découverte dynamique ou automatique, ou encore leur négociation et leur

composition pour former d'autres services sans une intervention humaine. Plusieurs opérations nécessitent la composition de services web, afin d'obtenir un service plus complexe — l'exemple le plus souvent cité dans ce document, i.e. l'organisation d'un voyage pour un client qui nécessite une location de voiture, une chambre d'hôtel ainsi qu'un billet d'avion, reflète cette nécessité de composition de services web.

Aucune norme utilisée par les services web ne permet cette composition automatique, puisqu'ils n'offrent pas une description suffisante des services. Il nous paraît alors nécessaire de se doter d'un algorithme qui réglerait ce problème de sémantique. Grâce à cette nouvelle approche, il sera possible de faire la recherche des services web et leur découverte non seulement par mots-clés, mais aussi par contexte ou domaine. Ainsi, la découverte des services web s'appuiera plus sur un raisonnement logique et sera axée d'une manière plus adéquate sur la satisfaction des besoins du client.

2.3.2.3 Problèmes de composition de services web

Les services web ont été conçus au départ pour automatiser la découverte ainsi que l'utilisation des services offerts sur le web. En effet, une application qui aurait besoin de connaître le taux de change d'une devise étrangère à une devise locale, devrait pouvoir chercher automatiquement dans un annuaire UDDI tous les sites donnant les taux de changes. Et ce, sans aucune intervention humaine.

Malgré l'attrait de cette idée, dans la pratique, de telles applications n'existent pas. Pourquoi ? Parce que même si les fournisseurs de services s'accordent à utiliser la même technologie, i.e. les services web, ils n'utilisent pas la même structure de données (différences syntaxiques), ni les mêmes unités (différences sémantiques). On parle ici d'une **barrière sémantique** qui complique la vie des utilisateurs des services web.

Prenons l'exemple suivant :

```
Fournisseur habituel de notre application
Ticket : getTicket(Origin ,Destination, Class, FlightCompany)
Avec  Ticket = {Time, Aeroport, Price}
      //Price in Eur
```

```
     Origin, Destination = {City, Aeroport}

Nouveau fournisseur
Ticket : getFlightTicket(Origin, Destination, FlightCompany)
Avec    Ticket = {Price, City, Aeroport}
        //Price in USD
        Origin, Destination = {Country, City, Aeroport}
```

Figure 2.3. Deux services web offrant le même service et ayant des données et des noms différents

Cet exemple illustre d'une manière éloquente notre problème de sémantique. En effet, nos deux fournisseurs offrent le même service : un billet d'avion vers la destination de notre choix. Mais, notre application étant habituée à la sémantique et à la signature de la première méthode (getTicket(Origin ,Destination, Class, FlightCompany)), elle sera incapable d'utiliser le service du nouveau fournisseur. On dit, dans ce cas, que notre application ne connaît qu'une seule *ontologie*, i.e. : elle ne dispose que d'un seul référentiel sémantique, celui de son fournisseur habituel. En particulier, les problèmes suivants retiennent notre attention :

- Les structures Ticket ont, certes, le même nom, mais ne contiennent pas les mêmes champs.

- Les deux champs Price ont le même nom, mais, encore une fois, ce n'est pas une garantie de leur égalité : dans un cas, il est affiché en Euros, dans l'autre cas, il est affiché en Dollars Américains.

- La notion de *synonymie* est une notion subjective. Deux signatures de méthodes différentes, donc deux termes différents, peuvent parfaitement vouloir dire la même chose : getTicket et getFlightTicket.

Ces problèmes créent une nécessité qu'on voudrait éviter : une intervention humaine. Aussi, la composition de services web est-elle nécessaire pour découvrir un service en particulier ou couvrir le besoin singulier d'un client. Mais qu'est-ce que la composition de services web ?

La composition de services web consiste en un rassemblement de plusieurs services web dans le but de combler un besoin précis. UDDI ne permet pas de faire ce rassemblement : il n'est pas possible avec cet outil de faire de la composition de services web, ni de faire leur adaptation de manière dynamique. En faisant de la composition de services web, on se propose de combiner des services — donc des fonctionnalités — ensemble et de les faire éventuellement interagir dynamiquement, selon nos besoins.

Notre travail de recherche s'attaque essentiellement à ce problème. Notre approche conçoit une manière de composition de services web fondée sur les types des éléments du WSDL, i.e. sur les documents d'affaire décrits dans les fichiers WSDL. En particulier, si nous faisons notre recherche en nous fondant sur le type des attributs, le problème de synonymie ne serait plus car on chercherait des opérations offrant des types précis sans faire cas de la signature de l'opération ou du nom qui lui est attribué. Aussi, le fait que les deux structures Ticket n'aient pas les mêmes attributs ne serait pas un obstacle : on chercherait par types (exemple : recherche des ensembles ayant les trois types (Time, Aeroport, Price)).

Nous décrivons cette approche et notre hypothèse de base dans le paragraphe suivant.

2.3.3. Problématique

Les services web connaissent ce qu'on appelle des problèmes d'interopérabilité : à cause des problèmes de sécurité, certaines compagnies mettent leurs propres dispositifs de sécurité. Ce qui empêche l'une d'utiliser les services web de l'autre. WS-I (*WS-Interoperability*) est un organisme qui cherche à coordonner toutes les initiatives dans ce domaine.

Et si ces problèmes n'existaient pas ? Dans la réalité de tous les jours, la complexité des développements impose très souvent l'utilisation de plusieurs services web pour parvenir au résultat souhaité. Dans une entreprise, un processus d'affaires contient de nombreux sous-processus. Comment, alors, composer ces services web pour atteindre le résultat qu'on souhaite ?

Quelques spécifications ont été définies pour aider les développeurs. Par exemple: BPEL4WS (*Business Process Execution Language For Web Services*), ou encore WSFL

(*Web Services Flow Language*). Grâce à ces outils et à la grammaire formelle qu'ils apportent, on peut décrire les processus d'affaires, les messages échangés, les opérations de composition de base, etc. Mais est-ce suffisant de composer des services web pour parvenir au résultat ou au service souhaité ?

2.4. Notre approche

2.4.1. Hypothèse de base

Nous partons d'un principe simple : Sur un document WSDL, les types des éléments sont des documents d'affaires et donnent une bonne idée du service qu'ils offrent. À partir de cette idée, nous croyons qu'en spécifiant les types des données en entrée et les types des données que nous escomptons atteindre, il est facile de trouver des réalisations, i.e. *l'enchaînement des services offerts dans un service web pour arriver aux données escomptées* (une définition plus formelle est décrite dans la section suivante).

Avec des langages comme C, il était toutefois difficile de trouver des résultats acceptables pour illustrer les réalisations qu'on voulait atteindre. Avec Java (et d'autres langages de programmation), les types sont plus riches et représentent des documents d'affaires. En effet, en langage C par exemple, faire une requête en demandant tous les services prenant en entrée deux *Strings* et retournant un *String* et un *Integer* aurait été peu satisfaisant : cette requête retournerait beaucoup de réalisations indésirables, i.e. réalisations qui prennent en entrée des données de même types que nous avons et qui retournent des données de mêmes types que ceux qu'on recherche, mais qui n'offrent pas le(s) service(s) escompté(s) (on s'y réfère plus tard en tant que « *false positives* »). Par contre, avec WSDL, les types représentent souvent des services (*FlightTicket*, *RoomPrice*, *CarReservation*, etc.), ce qui devrait réduire considérablement le nombre des « *false positives* ». Par conséquent, notre approche est basée sur un appariement de signatures pour désigner le service recherché. L'appariement de signatures est principalement un appariement de types. Nous détaillons cette approche dans le quatrième chapitre, et présentons les différents algorithmes utilisés pour l'appariement des signatures.

CHAPITRE 3

TECHNOLOGIES OU STANDARDS UTILISES POUR LES SERVICES WEB

Nous avons déjà évoqué les services web dans le chapitre précédent. En particulier, nous avons expliqué brièvement le fonctionnement d'un service web et présenté les protocoles liés aux services web.

Dans ce qui suit, nous présenterons le cycle de vie d'un service web, puis nous nous attarderons sur le protocole SOAP, le langage WSDL, la base de données UDDI, puis nous présenterons les avantages et les inconvénients des services web plus en détail.

3.1. Cycle de vie

L'emploi d'un service web connaît le cycle de vie suivant (Garcia et Henriet, 2004) :

- D'abord, on effectue le déploiement du service web, en fonction de la plate-forme.
- Ensuite, on enregistre le service web à l'aide de WSDL (*Web Services Description Language*), dans l'annuaire UDDI.
- L'étape suivante est la découverte du service web par le client, par l'intermédiaire d'UDDI qui lui donne accès à tous les services web inscrits. Pour ce, on utilise SOAP.

- Enfin, Le client invoque le service web voulu, ce qui termine le cycle de vie de ce service web.

Dans cette brève description du cycle de vie d'un service web, nous avons évoqué WSDL, UDDI et SOAP. Nous explorons chacun de ces standards dans ce qui suit.

3.2. SOAP (Simple Object Access Protocol)

SOAP est un protocole de communication pour l'échange d'informations dans un environnement décentralisé et distribué. *SOAP encourage le partage des données entre machines qui seront capables d'analyser facilement et correctement ces mêmes données* (Erenkrantz, 2002), grâce notamment à l'utilisation de XML comme principal format de données.

SOAP est composé de trois parties (W3C-SOAP) :

- Une enveloppe définissant un cadre pour décrire le contenu du message, comment y accéder et si c'est optionnel ou obligatoire.
- Un ensemble de règles de codage pour exprimer des instances de types de données définis pour une application.
- Une représentation dite RPC qui définit une convention pouvant être utilisée pour représenter des appels et des réponses à distance.

L'enveloppe et les règles de codage sont définies dans différents espaces de nom pour une meilleure simplicité par la modularité.

La figure suivante présente les deux messages SOAP possibles.

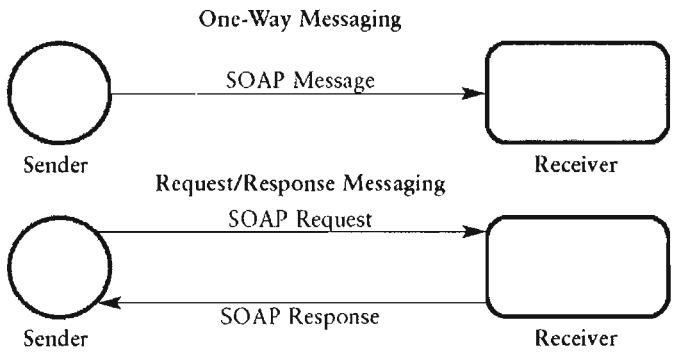


Figure 3.1. Messages SOAP

Nous expliquons dans ce qui suit la structure d'un message SOAP.

3.2.1. Structure d'un message SOAP

Un message SOAP est composé de (Hansen et al., 2002):

- Une enveloppe (***Envelope***) (Figure 3.2.): Considérée comme la racine d'un message SOAP, elle contient un attribut obligatoire, ***xmlns***, qui fait référence au schéma XML décrivant la structure du message SOAP (Cf. : W3Schools). Elle pourrait aussi contenir un autre attribut optionnel nommé ***encodingStyle***, qui définirait, s'il est présent, le style d'encodage du message SOAP. Cet attribut possède (W3C):
    - Un nom local (*local name*) du style d'encodage.
    - Un espace de nom (*namespace).*

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  ...
  Message information goes here
```

```
  ...
</soap:Envelope>
```

Figure 3.2. Illustration de l'élément ***Envelope*** *dans un document SOAP (c.f : W3schools)*

- Un en-tête optionnel (*Optional Header*) : Contient des informations spécifiques (authentification, payement, etc.) au message SOAP. S'il est présent, il doit être le premier « enfant » (*child element*) de l'élément *Envelope* (Figure 3.3.).

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header>
<m:Trans
xmlns:m="http://www.w3schools.com/transaction/"
soap:mustUnderstand="1">234</m:Trans>
</soap:Header>
...
...
</soap:Envelope>
```

Figure 3.3. Illustration de l'élément Header dans un document SOAP (Cf. : W3schools)

- Un élément corps obligatoire (*Mandatory Body*) : Contient le message réel destiné au point final du message (Figure 3.4.

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body>
   <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
      <m:Item>Apples</m:Item>
   </m:GetPrice>
</soap:Body>
</soap:Envelope>
```

*Figure 3.4. Illustration de l'élément **Body** dans un document SOAP (Cf. : W3schools)*

La Figure 3.5. figure suivante présente les différents composants d'une requête SOAP et la façon dont ils y sont classés.

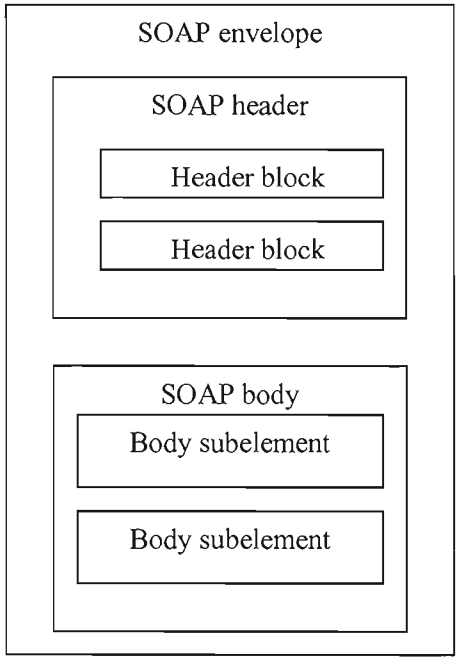


Figure 3.5. La structure d'un message SOAP (inspiré d'Apple)

3.2.2. Gestion d'une exception dans un message SOAP

Si un message SOAP contient une requête qui ne peut être traitée pas son récepteur, celui-ci retourne un message d'erreur (Figure 3.6.) sous format XML expliquant la nature du problème.

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-
ENV:Client.AuthenticationFailure</faultcode>
         <faultstring>Failed to authenticate client</faultstring>
         <faultactor>urn:X-SkatesTown:PartnerGateway</faultactor>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3.6. Message d'erreur pour requête contenant une erreur

3.2.3. Patrons d'échange de messages SOAP

Le protocole SOAP comporte quelques patrons d'échange de messages SOAP :

- *Message de type requête-réponse* : Le client envoie une requête SOAP au serveur et attend une réponse.

- *Message à sens unique* : La requête est envoyée au serveur sans qu'aucune réponse ne soit attendue.

- *Message de type notification* : Le serveur envoie un message SOAP au client pour notification.

- *Message sollicité* : Le serveur envoie un message SOAP au client et reçoit une réponse de ce dernier.

Ces patrons d'échange sont illustrés dans la figure suivante.

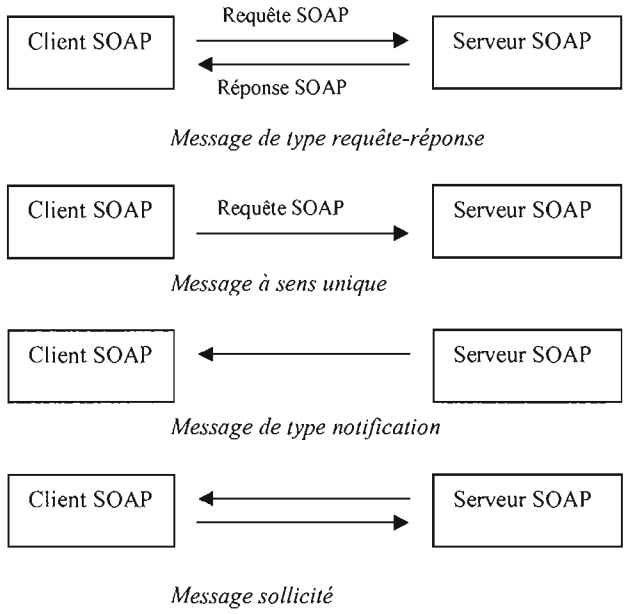


Figure 3.7. Patrons d'échange de messages SOAP

3.2.4. Le transporteur idéal de SOAP

Il n'est indiqué nulle part dans le protocole SOAP quel transporteur il faudrait utiliser. Cependant, *HTTP* est tellement populaire qu'il devient le choix numéro un. Le transport de SOAP sur *HTTP* ne nécessite aucune reconfiguration du pare-feu de la part des administrateurs réseau : les ports 80 et 8080, qui sont utilisés par *HTTP,* sont le plus souvent ouverts sur les pare-feu.

La version 1.1 du protocole SOAP indique que l'échange *HTTP* est fondé sur des requêtes de types POST. Nous présentons dans la Figure 3.8. un exemple d'une requête

SOAP via *HTTP* utilisant POST où on demande le prix de la dernière transaction (*GetLastTradePrice*) .

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3.8. Requête SOAP/http utilisant POST (W3C, 2000)

- Le **Content-Type** indique que le corps de la requête est une présentation XML.
- Le **SOAPAction** indique la destination de la requête SOAP.

La réponse à la requête SOAP/HTTP indiquera si elle a finalement été traitée correctement ou pas. La Figure 3.9. illustre la réponse à la requête de la Figure 3.8. (Le prix est de 34.5).

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3.9. Réponse d'une requête SOAP/http (W3C, 2000)

SOAP est considéré comme le protocole standard des services web. Toutefois, il présente certains inconvénients. Son principal défaut est sa faiblesse au niveau de la sécurité. En effet, un message SOAP est assez facile à décoder et éventuellement à modifier puisqu'il est encodé en mode texte.

L'extension WS-Security (IBM-WS-Security, 2002) permet de placer des informations de sécurité de base dans les en-têtes *(headers)* d'un message SOAP de la manière suivante (Atkinson et al., 2002).

On ajoute des blocs de sécurité grâce à l'en-tête *<Security>*, chacun pour un récepteur précis. Un seul en-tête peut ne pas avoir l'attribut *S: actor*. Tous les autres ont cet attribut, de manière que chaque *S: actor* est différent d'en-tête en en-tête. Deux messages pour deux récepteurs différents doivent apparaître sous des en-têtes différents. L'en-tête qui ne contient pas l'attribut *S: actor* peut être consommé par n'importe quel récepteur. La Figure 3.10. illustre un exemple d'enveloppe contenant un en-tête de sécurité.

```
<S:Envelope>
<S:Header>
...
<Security S:actor="..." S:mustUnderstand="...">
...
</Security>
...
</S:Header>
```

```
...
</S:Envelope>
```

Figure 3.10. Syntaxe d'un en-tête contenant un élément de sécurité (Atkinson et al.)

Le protocole SOAP ne décrit que la façon d'échanger des données à l'aide d'un service web. C'est grâce au standard WSDL que ces descriptions sont faites. Dans la section suivante nous parlons plus amplement de ce standard.

3.3. UDDI (Universal Description Discovery and Integration)

UDDI est un annuaire de type XML, qui permet de bâtir des annuaires des entreprises e-business, ainsi que leurs produits et services, leur permettant de s'enregistrer et de publier les services web qu'elles désirent offrir au grand public (OASIS-UDDI, 2002). L'idée derrière UDDI est de normaliser le format des entrées d'entreprise et de services dans un annuaire pour faciliter la découverte des services web et encourager les échanges d'affaires entre elles (B2B).

UDDI sert de :

- Pages jaunes (*Yellow Pages*) : il indique ce que fait le service.
- Pages blanches (*White Pages*) : il donne des informations sur le fournisseur du service.
- Pages vertes : il donne des détails sur comment utiliser techniquement le service web en question (UDDI, 2002).

Le modèle d'information utilisé par les registres UDDI est défini dans un schéma XML. Ce schéma définit pour chaque service web quatre types d'informations : l'information d'affaires (*business information*), l'information sur le service (*service information*), l'information sur les liens (*binding information*) et l'information sur les spécifications des services (*information about specifications for services*) (Qi et al., 2004).

Le registre UDDI utilise le protocole SOAP ainsi que le protocole HTTP, et suit un modèle d'échange de données fondé sur la notation XML.

La figure suivante montre bien ce mécanisme.

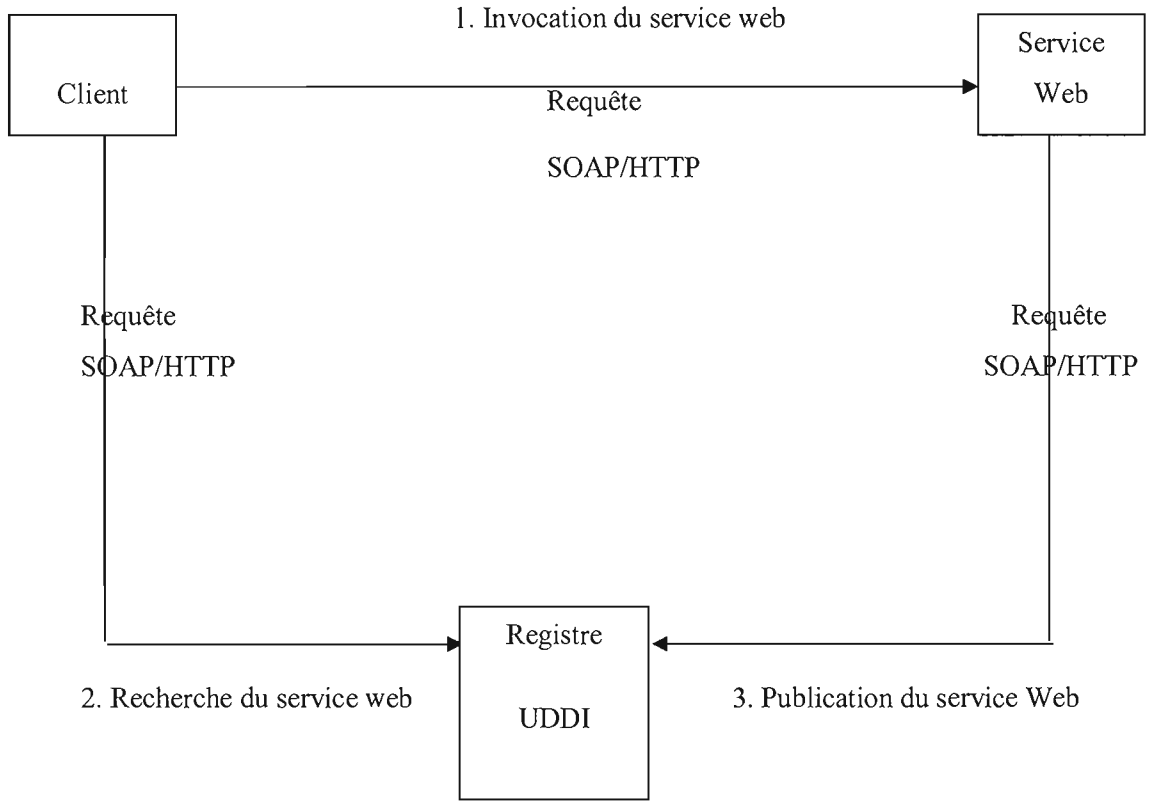


Figure 3.11. Publication dans un registre UDDI

La Figure 3.12. présente un meta-modèle simplifié des entrées UDDI. L'entrée principale est l'entité d'affaires (*BusinessEntity*) qui contient des descriptions abstraites du service d'affaires (*BusinessService*), qui sont à leur tour liées à des mises en œuvre concrètes (*BindingTemplate and TechnicalModel*).

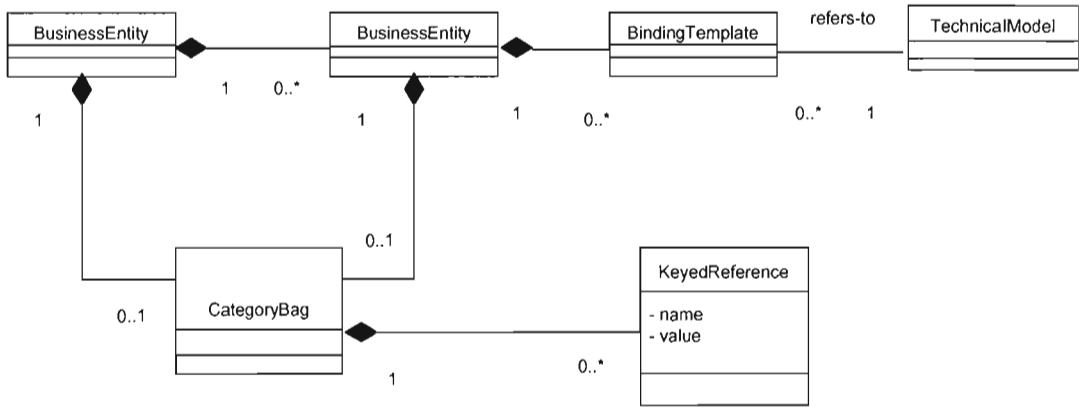


Figure 3.12. Un méta-model simplifié d'entrées UDDI (Mili et al., 2004)

3.4. Avantages et inconvénients des services web

3.4.1. Pourquoi choisir les services web?

De nos jours, il est devenu primordial, dans le domaine du développement logiciel, d'intégrer plusieurs composantes réparties dans plusieurs sites/lieux, à l'aide d'outils souvent hétérogènes, pour atteindre les résultats dont on a besoin. Les services web permettent justement cette intégration, ce qui explique leur popularité. Avant de présenter en détail les avantages des services web (section 3.4.2.3), nous faisons une introduction dans ce qui suit.

3.4.2. WSDL (Web Service Definition Language)

3.4.2.1 Introduction

WSDL est un langage de description de service web sous format XML (W3C – WSDL, 2002), qui a été initialement proposé par IBM et Microsoft en 2001. Il permet de définir la structure des objets d'une manière abstraite et indépendante du langage de développement, et ce quels que soient le protocole (SOAP ou XML) ou le codage (MIME) utilisés par ces objets spécifiques à chaque système (Hansen et al., 2002). Un fichier WSDL comporte les éléments suivants :

- Définitions abstraites des données qu'on souhaite transmettre.

- Informations de type *adresse* afin de localiser le service à spécifier.
- Informations sur le type de données pour tous les messages XML.
- Informations sur toutes les fonctions disponibles publiquement.
- Informations obligatoires sur le protocole de transport qu'il faut spécifiquement utiliser.

WSDL propose une double description du service :

- Une vue dite *abstraite* qui présente les opérations et les messages des services.
- Une vue dite *concrète* qui présente les choix de mise en œuvre faits par le fournisseur du service.

La figure suivante illustre le déroulement d'une consultation de document WSDL.

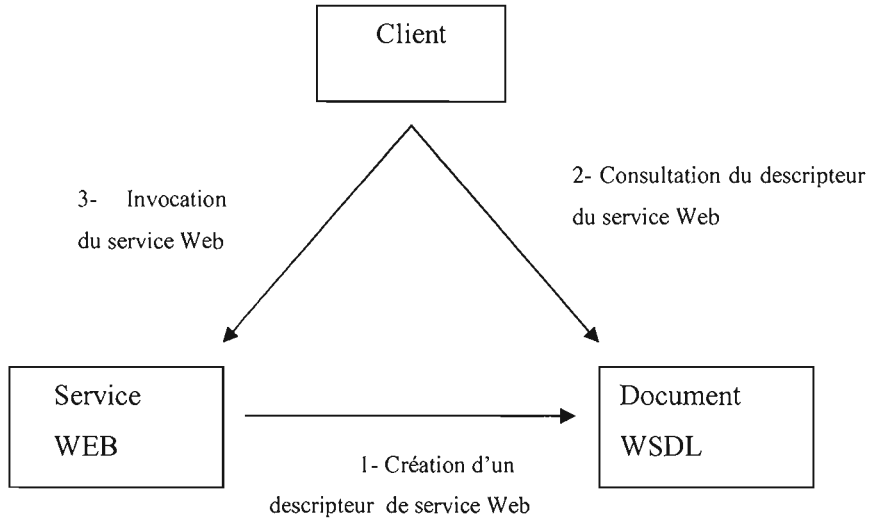


Figure 3.13. Déroulement d'une consultation d'un document WSDL

3.4.2.2 WSDL en détail

Un document WSDL est un fichier XML qui définit des services comme une collection de points finaux (*endpoints*) de réseau ou des **ports** (W3C). Il décrit en détail comment utiliser un service web. Il est actuellement à sa version 2.0.

Structure d'un fichier WSDL

Dans un fichier WSDL, chaque service web est une entité (*entity*), qui est définie par des ports (*ports*), qui sont les *points finaux (endpoints)* du service. Ces mêmes ports peuvent recevoir et répondre à un ensemble de messages définis par le type du port (*port's type*). Chaque port est, en réalité, la liaison entre le *portType* et un protocole d'accès, qui nous dit comment les messages doivent être encodés et envoyés au port.

La Figure 3.14. présente un exemple d'un fichier WSDL.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>
   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
```

```
    </operation>
  </portType>

  <binding name="Hello_Binding" type="tns:Hello_PortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
        <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </input>
      <output>
        <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </output>
    </operation>
  </binding>

  <service name="Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address
       location="http://localhost:8080/soap/servlet/rpcrouter"/>
    </port>
  </service>
</definitions>
```

Figure 3.14. Exemple d'un fichier WSDL

Dans ce qui suit, nous définissons chacun des éléments mis en évidence dans la Figure 3.14.

Définitions : Cet élément (Figure 3.15.) est considéré comme la racine du document WSDL. Il définit le nom du service web, déclare les différents espaces de nom utilisés dans les documents et contient tous les éléments décrits ici.

```
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote/service"
          xmlns:tns="http://example.com/stockquote/service"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:defs="http://example.com/stockquote/definitions"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
```

*Figure 3.15. L'élément **definition** du fichier WSDL (W3C)*

Types : Cet élément (Figure 3.16.) contient des types de données importants pour les messages échangés. Pour un maximum d'interopérabilité et une neutralité de la plateforme, WSDL utilise XSD comme *canonical type system.*

```
<definitions  name="nmtoken"? targetNamespace="uri"? >
    <types>
        <xsd:schema .... />
    </types>
</definitions>
```

*Figure 3.16. L'élément **types** du fichier WSDL (W3C)*

Message : Il représente une définition abstraite des données transmises et consiste en un ou plusieurs **parts** logiques (*logical parts*) (Figure 3.17.). Il peut être un message d'entrée ou un message de sortie d'un service web.

```
<definitions .... >
    <message name="nmtoken"> *
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </message>
</definitions>
```

*Figure 3.17. L'élément **message** d'un fichier WSDL (W3C)*

PortType : Un service web peut contenir plusieurs portTypes. Chaque portType définit un ensemble d'opérations abstraites offertes par le service web. Un portType est identifié par un nom et contient une opération au moins. Chaque opération a un nom et contient au plus un message d'entrée – défini par l'élément ***input*** – et au plus un message de sortie – défini par l'élément ***output***. En cas d'exception, l'élément **fault** est présent et définit un message d'erreur retourné.

La Figure 3.18. illustre tous ces différents éléments.

```
<wsdl:definitions .... >
    <wsdl:portType .... > *
        <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
           <wsdl:input name="nmtoken"? message="qname"/>
           <wsdl:output name="nmtoken"? message="qname"/>
           <wsdl:fault name="nmtoken" message="qname"/>*
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

*Figure 3.18. L'élément **portType** d'un fichier WSDL (W3C)*

Quatre types d'opérations sont décrits dans la spécification WSDL (W3C).

- Opération à **sens unique** (Figure 3.19.) : reçoit un input et n'envoie rien du tout.

```
<wsdl:definitions .... > <wsdl:portType .... > *
        <wsdl:operation name="nmtoken">
           <wsdl:input name="nmtoken"? message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

Figure 3.19. Exemple d'une opération à sens unique (W3C)

- Opération **requête-réponse** (Figure 3.20.): reçoit un *input* et envoie un *input*.

```
<wsdl:definitions .... > <wsdl:portType .... > *
        <wsdl:operation name="nmtoken">
           <wsdl:input name="nmtoken"? message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

Figure 3.20. Exemple d'une opération requite-réponse (W3C)

- Opération **sollicitant une réponse** (Figure 3.21.) : envoie un *output* et reçoit un *input*.

```
<wsdl:definitions .... > <wsdl:portType .... > *
        <wsdl:operation name="nmtoken">
           <wsdl:input name="nmtoken"? message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

Figure 3.21. Exemple d'une opération sollicitant une réponse (W3C)

- Opération **avec notification** (Figure 3.22.): envoie un *output* mais ne reçoit pas d'*input*.

```
<wsdl:definitions .... >
    <wsdl:portType .... > *
        <wsdl:operation name="nmtoken">
           <wsdl:input name="nmtoken"? message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

Figure 3.22. Exemple d'une opération avec notification (W3C)

Binding : Il définit le format d'un message et les protocoles de communication pour des opérations et des messages définis par un *portType* (Figure 3.23.). Un fichier WSDL peut contenir plusieurs éléments *Binding* dont chacun est identifié par un nom et fait référence à un *portType*.

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *
        <-- extensibility element (1) --> *
        <wsdl:operation name="nmtoken"> *
            <-- extensibility element (2) --> *
            <wsdl:input name="nmtoken"? > ?
                <-- extensibility element (3) -->
            </wsdl:input>
            <wsdl:output name="nmtoken"? > ?
                <-- extensibility element (4) --> *
            </wsdl:output>
            <wsdl:fault name="nmtoken"> *
                <-- extensibility element (5) --> *
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

*Figure 3.23. L'élément **binding** d'un fichier WSDL (W3C)*

Service : Regroupe un ensemble de ports reliés entre eux. Il a un nom et contient un ou plusieurs ports. Chaque élément port a un nom et définit un point d'accès au service web (Figure 3.24.).

```
<service name="StockQuoteService">
     <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
            <soap:address location="http://example.com/stockquote"/>
        </port>
</service>
```

*Figure 3.24. L'élément **service** d'un fichier WSDL (W3C)*

L'élément **import** ne figure pas dans la Figure 3.14. , mais il est nécessaire de le mentionner : on l'utilise pour importer d'autres documents WSDL en associant un espace de nom à la location d'un document (Figure 3.25.).

```
<definitions .... >
    <import namespace="uri" location="uri"/> *
</definitions>
```

Figure 3.25. L'élément import d'un fichier WSDL (W3C)

Un document WSDL décrit comment utiliser un service web. Mais comment trouver ce service web ? Comment localiser le document WSDL qui le décrit ? Le registre UDDI est la réponse à cette question. On le présente dans le paragraphe suivant.

3.4.2.3 Avantages des services web

Grâce aux services web, il est possible de diviser des applications et des processus d'affaires en plusieurs *services* qui effectuent, chacun, une tâche précise. L'interopérabilité s'en trouve facilitée puisque ces services peuvent être utilisés aussi bien à l'intérieur qu'à l'extérieur d'une entreprise.

Les services web permettent à des portions de logiciels écrits dans des langages différents ou évoluant sur différents systèmes d'exploitation, de communiquer facilement entre elles. Des applications supportant différents processus d'une ou plusieurs organisations peuvent, aussi, communiquer entre elles et s'échanger des données grâce aux services web (Babin et Leblanc, 2003).

Les services web offrent aux entreprises la flexibilité de réponse et d'anticipation des besoins — toujours en évolution — des clients, la flexibilité d'interaction et de configuration avec les partenaires externes, ainsi que la rationalisation des infrastructures logicielles et la baisse de coût de leur maintenance. Pour ce faire, les services web offrent les avantages suivants aux entreprises (Babin et Leblanc, 2003).

- L'information dont un client aurait besoin, pour interagir avec une entreprise, ainsi que toute donnée ou fonctionnalité pertinente, sont publiques et à sa portée.

- Les partenaires d'une entreprise, qui ont un ou plusieurs clients en commun avec la dite entreprise, sont munis de toute information ou fonctionnalité nécessaire pour servir ce client : la fonctionnalité est publique.

- Les équipes de développement peuvent travailler indépendamment l'une de l'autre, sur des systèmes qui, potentiellement, auront à interagir ensemble. Le travail synchronisé, parfois assez coûteux, n'est donc plus nécessaire.

- Les fournisseurs d'une entreprise peuvent facilement ajuster leur inventaire en fonction de ses besoins : ils ont un accès direct à l'information nécessaire pour cette tâche.

Les entreprises œuvrant dans l'e-commerce font face à un autre défi, le nombre croissant des partenaires potentiels. Gérer cette croissance n'est pas un exercice facile. D'abord, cette croissance est assez complexe à construire. Mais surtout, il est assez coûteux de déployer des interfaces entre les systèmes d'information des dits partenaires. Les services web, en réponse à ce problème de poids, proposent les solutions suivantes (Babin et Leblanc, 2003).

- Simplicité : Contrairement à l'architecture client/serveur, les services web permettent de créer la fonctionnalité une seule fois, plutôt que d'obliger les participants à la reproduire à chacun des bouts. On dit que les services web *réduisent la complexité des branchements* (Babin et Leblanc, 2003).

- Ouverture : Les services web permettent de tirer profit des infrastructures informatiques existantes et des plateformes ouvertes (Internet, par exemple). Les « *lock-in* » que les entreprises subissaient traditionnellement de la part de leurs fournisseurs informatiques ont, aussi, moins d'impact puisque tous les services sont offerts publiquement.

- Hétérogénéité : Elle importe peu avec les services web. Ils décrivent comment transmettre un message normalisé entre deux applications, sans pour autant imposer une manière de le construire.

- Faible couplage et réutilisation : Le faible couplage des interfaces associées aux services web, ainsi que leur architecture modulaire, permettent la réutilisation des services au sein d'autres modules.

3.4.3. Les inconvénients

Les services web souffrent d'un certain nombre de faiblesses. Nous en citons quelques unes dans ce qui suit.

3.4.3.1 Problèmes de fiabilité

Dans un domaine qui demande une certaine connexion entre différents services provenant de différents lieux/serveurs, les erreurs sont probables. Il s'agit de les éviter le plus possible, et de donner de l'information sur la réussite ou pas de la tentative d'atteinte du service quand elles se produisent. Les erreurs de syntaxe ou les requêtes mal formulées ne sont pas un véritable problème : il existe assez d'outils pour s'en occuper. Le principal souci de fiabilité dans le domaine des services web réside plutôt dans la livraison des services d'infrastructure, i.e. la livraison de messages, la découverte de services, les comportements fiables entre les services eux-mêmes, ainsi que les comportements fiables du demandeur et du fournisseur. Ces problèmes sont considérés séparément des autres problèmes de sécurité ou de disponibilité. Ils peuvent néanmoins se chevaucher et se produire en même temps (Cf. : W3C).

Nous allons discuter de deux aspects de la fiabilité : la fiabilité du message et la fiabilité du service. Nous ne discuterons pas des actes délibérés qu'une personne peut causer à la sécurité des services web — tels que les attaques ou le piratage — c'est un domaine assez vaste qui ne touche pas uniquement les services web.

Fiabilité du message

Quand il s'agit d'envoyer un message, deux pratiques nous intéressent en ce qui concerne la fiabilité du message : a-t-il été reçu ou pas ? Si oui, a-t-il été reçu une seule fois ou plusieurs fois ? Car savoir si un message n'a pas été reçu nous poussera, au moins, à l'envoyer de nouveau. Recevoir le même message plusieurs fois, par contre, peut donner un

indice sur un défaut au niveau du mécanisme d'envoi des messages. Maintenant quand un message est reçu, trois questions se posent sur sa fiabilité (Babin et Leblanc, 2003):

- Est-ce que le message reçu est le même que celui qui a été envoyé ? Une telle vérification peut être faite grâce à des techniques répandues telles que le total de contrôle (*check-sum*), le *bit-counting* ou les signatures digitales. Le risque sera, néanmoins, toujours là.

- Est-ce que le message reçu respecte les formats préétablis dans le protocole de messagerie ? Cette vérification concerne l'aspect de la syntaxe (est-ce que le code XML est conforme ?), ainsi que celui de la structure (validation avec d'autres schémas XML ou des messages de définition de WSDL).

- Est-ce que le message respecte les règles d'affaires du correspondant qui le reçoit ? Cette validation est faite par une combinaison de vérifications syntaxique et structurale faites par des outils automatisés ou par des humains spécialisés dans ce genre de vérification.

La fiabilité des messages est souvent atteinte grâce à une infrastructure de reconnaissance, c'est-à-dire un ensemble de règles qui décrivent comment les parties concernées par un message peuvent communiquer entre elles au sujet de sa réception et de sa validité. *WS-Reliability* et *WS-ReliableMessaging* sont de bons exemples de spécifications d'une infrastructure de reconnaissance qui accroissent le niveau d'extensibilité de SOAP (Cf. : W3C).

Fiabilité du service

Il est impossible de garantir que les fournisseurs et les demandeurs de services se comporteront toujours d'une manière parfaite. On peut néanmoins établir des stratégies ou des techniques pour maximiser cette fiabilité et réduire les risques d'échec. La principale technique utilisée est celle appelée *gestion transactionnelle de contexte*, qui permet la communication entre les différentes parties impliquées dans l'utilisation d'un service. Grâce

à cette communication, ces parties ont une constante rétroaction (*feedback*) sur l'état de la transaction et en cas d'échec, les transactions peuvent être annulées ou refaites de nouveau.

L'approche-clé utilisée dans le contexte des services est le déploiement de services dits services-tiers (*third party services*), qui surveillent les services et veillent à ce qu'ils soient utilisés dans des rôles spécifiques, dont les demandeurs et les fournisseurs ne se soucient pas. Chacun de ces services-tiers établit une fonctionnalité spécifique, souvent codée dans des entêtes spécifiques des messages (Cf. : W3C).

3.4.3.2 Problèmes légaux

Les services web étant encore une technologie relativement nouvelle, plusieurs questions de nature légale se posent. En particulier, si X et Y sont deux entreprises utilisant des services web et effectuant des transactions ensemble (Babin et Leblanc, 2003).

- Quelle est la nature de la relation entre X et Y ? Quel est le niveau de cohérence et de couplage entre X et Y (léger ou fort) ?
- Quel est le niveau de confiance entre X et Y ?
- Qu'est-ce qui appartient à qui dans le contexte d'échange de services web ?
- Comment s'assurer de la qualité d'un service web par un processus commun et « neutre » ?
- Quels sont les risques associés à la responsabilité de X et de Y dans une de leurs transactions ? Partagent-ils des risques ? Qui est imputable d'une responsabilité ? Sur quoi se baserait-on pour imputer cette responsabilité à X ou à Y ?

Ces questions se posent chaque jour. Certaines trouvent des réponses naturelles et intuitives. D'autres auront certainement besoin d'un cadre régisseur : des forums internationaux, des entreprises ou des organismes de normalisation.

3.4.3.3 Conclusion

Les services web montrent certes quelques lacunes de fiabilité, et en sont encore à leur balbutiement juridique, mais ce sont encore des technologies assez nouvelles qui présentent des avantages intéressants et offrent des solutions aux problèmes d'échanges de données et d'intégration d'applications. Pour découvrir des services précis et répondre aux besoins du client, il est parfois primordial de composer plusieurs services web. Dans le chapitre qui suit, nous allons discuter plus en détail de la composition de services web qui a connu beaucoup d'intérêt dans les dernières années.

CHAPITRE 4

LA COMPOSITION DE SERVICES WEB

4.1. Utilisation de la composition de services web

La recherche fondée sur le contenu des services web, via le standard UDDI, montre des lacunes dans trois domaines reliés entre eux d'une façon ou d'une autre :

D'abord, les entités d'affaires, ainsi que les services d'affaires ne peuvent être recherchés que par des champs textuels non-normalisés (exemple : les noms des services) ou des champs semi-normalisés : ce qu'on appelle les références indexées, i.e. tout ce qui correspond à des facettes de classification (*classification facets*) utilisées dans la recherche d'information (Salton et McGill, 1983) et dans le contexte de bibliothèques de code (*software librairies*) (Poulin et al., 1993). Cependant, cette classification souffre de quelques imperfections, d'abord, elle demande une certaine familiarité et connaissance avec les taxonomies ou ontologies correspondantes. Ensuite, elle repose sur la classification parfois laborieuse de l'être humain, puisque elle ne dépend pas de la sémantique intrinsèque du service.

Le deuxième problème est celui qui nous intéresse ici : la composition de services web. Même si on peut interroger une base de données en nous fondant sur la sémantique des descriptions fournies par WSDL – par opposition aux références indexées dont nons avons parlé ci-dessus – les requêtes supportées par UDDI cherchent un simple service qui satisfait

un besoin d'affaires, et non pas une composition de services qui satisfait un besoin d'affaires. Nous illustrons ce problème par notre exemple du voyageur et son forfait : considérons le cas d'un voyageur qui aurait besoin d'un forfait-voyage incluant un billet d'avion aller-retour, une chambre d'hôtel et une voiture de location à la destination choisie.

Ce voyageur doit chercher un service à la fois et les invoquer ultérieurement. Il serait plus facile et plus rapide de faire une seule recherche en fournissant les données voulues au départ et en recevant au retour toutes ces données en même temps.

Le troisième problème avec UDDI, et non pas le moindre, est le suivant : les services d'affaires n'ont pas besoin d'être techniquement décrits en utilisant un langage de description normalisé, car UDDI laisse cette description ouverte et sans restrictions. Le modèle dit technique *peut* se référer à la spécification de WSDL, mais il n'est pas du tout *obligé* de le faire. Par conséquence, il nous est impossible de faire une recherche fondée sur une description de web services dont le modèle de représentation est uniforme.

Il n'est pas de notre ressort de trouver une solution pour ce dernier problème. Nous accepterons durant notre travail les limitations conséquentes d'une description des services web fondée sur une spécification WSDL.

4.2. Classification des approches de composition de services web

La découverte (ou invocation) des services web a connu un certain intérêt lors des dernières années. En particulier, plusieurs recherches ont été réalisées sur la composition de services web et la façon dont cette technique pourrait être utilisée pour arriver à certains résultats. Paolucci et al., (2003) ainsi que Colgrave et al., (2004) se sont intéressés à rendre l'invocation et la composition de services web plus flexible. D'autres chercheurs se sont penchés sur la Qualité de Service (QoS) pour raffiner la recherche des services web.

La composition de services web nécessite un modèle abstrait de description d'un flux des services. Les approches proposées dans la littérature sont diverses. Aucune n'est

reconnue comme une norme. La classification de ces approches nous permet de dégager trois grandes familles (Ter Beek et al., 2006).

- L'approche sémantique : se base essentiellement sur l'ontologie OWL-S pour décrire les services composés.
- L'approche industrielle : propose des langages de formalisation basés sur XML.
- L'approche formelle : utilise des techniques de modélisation de validation formelle de processus (exemple : les réseaux de Petri).

Toutes ces approches nous permettent de décrire d'une manière abstraite le flux de services web à composer. Ensuite, il suffit d'appliquer des algorithmes de composition de services web et construire ainsi ces flux dynamiquement. La composition dynamique de services web se fonde essentiellement sur l'intelligence artificielle et ses techniques de planification. Dans ce qui suit, nous décrivons les trois approches de composition de services web citées ci-haut, avant de présenter les approches et algorithmes proposés pour la composition dynamique de services web.

4.2.1. Composition de services web : approche sémantique (OWL-S)

Pour explorer l'approche sémantique de composition de services web, la communauté académique a proposé au début DAML-S. L'idée était d'utiliser un langage formel afin de définir ce qu'un service web fait (Cardoso et al., 2005), puisque WSDL est plus axé sur l'aspect opérationnel, ce qui le rend insuffisant pour la découverte et la composition de services web. C'est ainsi qu'on a rajouté

i) un profil, qui décrit *ce que fait un service web*

ii) un processus, qui décrit *comment ça marche*,

iii) des descriptions (*grounding descriptions*) qui font la correspondance ces composants au WSDL (OWL-S, 2004).

Aujourd'hui, OWL-S a remplacé DAML-S (même communauté). On utilise l'ontologie OWL-S pour décrire les flux composés de services web en *se basant sur l'utilisation des techniques de planification de l'intelligence artificielle, afin de chercher, orchestrer, composer et exécuter les services* (Zhang et Zhang, 2006), OWL-S étant une ontologie de services web basée sur OWL, qui *offre aux fournisseurs des services web un ensemble noyau de balises permettant de décrire les propriétés de ces services web* (OWL-S, 2004).

OWL-S ne constitue pas la seule possibilité quand il s'agit d'approche sémantique de composition de services web : Bleul et al., (2007) proposent trois algorithmes fondés sur l'approche sémantique. L'algorithme *IDDFS* - gagnant du Web Service Challenge 2006, WSC'06 - qui *est spécialement rapide à trouver des solutions si le répertoire des services web est de taille petite ou de moyenne* (Bleul et al., 2007); l'algorithme *heuristique* qui convient pour un répertoire de services web plus grand; et l'algorithme *génétique* approprié aux répertoires de services web de taille particulièrement grande. Le choix entre les trois algorithmes est fait grâce à un planificateur stratégique utilisé à chaque demande de compositions. Bleul et al., (2007) ont aussi proposé une interface de services web en incluant différents formats de documents d'import et d'export comme XSD, OWL et OWL-S, afin d'avoir un système utilisable par toutes les approches de gestion de services se fondant sur la sémantique pour découvrir des services web.

4.2.2. Composition de services web : approche industrielle (BPEL4WS)

BPEL4WS (*Business Process Execution Language for Web Services*) offre un langage pour les spécifications formelles des processus d'affaires et leurs interactions (BPEL4WS). IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems ont tous contribué à l'élaboration de BPEL4WS.

BPEL4WS supporte la modélisation de deux types de processus : les processus exécutables et ceux abstraits. Un processus abstrait est un protocole d'affaires qui spécifie le comportement des échanges de messages entre plusieurs composantes, sans pour autant réveler leurs comportements internes. Un processus exécutable spécifie l'ordre d'exécution

entre les activités (*activities*) constituant le processus, les partenaires (*partners*) du processus, les messages échangés entre ces partenaires, et indique le comportement en cas d'erreur ou d'exception (Wohed et al., 2002).

Dans l'exemple de la Figure 4.1, nous présentons un exemple d'un document BPEL4WS composant deux services, l'un vérifiant la disponibilité d'une chambre dans un hôtel et l'autre faisant la réservation d'une chambre d'hôtel :

```
<process name="HotelReservation">
<partnerLinks>
        <partnerLinks name = "RoomChecker">
        <partnerLinks name = "RoomBooker">
</partnerLinks>

(...)

<sequence>
        <sequence>
                <invoke partnerLink = "RoomChecker"
                        Operation = "RoomAvailability"
                        inputVariable = "HotelId" />

                <receive partnerLink = "RoomChecker"
                        Operation = "RoomAvailability"
                        variable = "HotelResponse" />
        </sequence>

        <switch>

                <case condition= "getVariableData('HotelResponse')== true">

                <sequence>

                <invoke partnerLink = "RoomBooker"
                        Operation = "BookRoom"
                        inputVariable = "HotelId" />

                <receive partnerLink = "RoomChecker"
                        Operation = "BookRoom"
                        inputVariable = "BookingResponse" />
                </sequence>
        </switch>
```

```
        </sequence>
</process>
```

Figure 4.1. Exemple d'un document BPEL4WS présentant une composition de services

4.2.3. Composition de services web : approche formelle

Le but de l'approche formelle de composition de services web est d'offrir un mécanisme de validation formelle des services web composés. Les chercheurs prônant cette approche se basent, en général, sur une modélisation sous forme de réseaux de Petri (Jeng, 1997), un modèle de formalisation de processus.

Un réseau de Petri est un graphe dirigé, connecté, où chaque nœud est ou bien une place ou bien une transition. Des jetons occupent les places. Quand il y a au moins un jeton dans chaque place connectée à une transition, nous parlons d'une transition activée (*enabled*). Suite à cette activation, un jeton est enlevé de chaque place en amont (*input place*) et mis dans une place en aval (*output place*) (Hamadi et al., 2003). *Un service web est modélisé sous forme de réseau de Petri en associant les opérations aux transitions et les états de service en places* (Hamadi et al., 2003). Ensuite, pour composer des services web, il suffit d'appliquer des opérateurs de composition aux réseaux de Petri représentant les dits services web (chaque service web est représenté par un réseau de Petri). Un réseau de Petri avec une place d'entrée (*input place*) pour recevoir des informations et une place de sortie (*output place*) pour émettre des informations, facilite la définition des opérateurs de composition, ainsi que l'analyse et la vérification de certaines propriétés comme l'accessibilité et l'absence de cycles.

Les réseaux de Petri ne constituent pas la seule approche formelle essayée. Bauer et Huget, (2004), ainsi que Skogan et al., (2004) ont exploré des approches se basant sur UML. Dans cette approche, on modélise la composition de services web sous forme de diagramme d'activités.

4.2.4. Conclusion

Ces approches sont intéressantes et ont ouvert d'importantes voies de recherche. Néanmoins, elles sollicitent des outils souvent peu compatibles entre eux-mêmes ou avec les normes. L'avantage de notre approche, c'est qu'elle ne nécessite aucun outil, aucune

modification des standards (UDDI, SOAP, etc.), et est, de ce fait, parfaitement utilisable et envisageable. Nous partons d'un algorithme que nous pouvons programmer avec le langage Java. Notre approche est, pour ainsi dire, dépourvue de tout effort de migration de données ou de mise à jour d'un quelconque standard.

CHAPITRE 5

COMPOSITION DE SERVICES PAR COUVERTURE DE FONCTIONS

Dans le chapitre trois, nous avons évoqué trois domaines dans lesquelles UDDI montre des lacunes majeures :

1. les entités d'affaires, ainsi que les services d'affaires, ne peuvent être recherchés que par des champs textuels non-normalisés ou des champs semi-normalisés.

2. les services d'affaires n'ont pas besoin d'être techniquement décrits en utilisant un langage de description standard, car UDDI laisse cette description ouverte et sans restrictions.

3. les requêtes supportées par UDDI cherchent un simple service qui satisfait un besoin d'affaires, et non pas une composition de services qui satisfait un besoin d'affaires.

Dans ce travail, nous essayons de trouver une réponse au troisième problème. Pour ce faire, nous composons des services web, en utilisant les références indexées d'UDDI et les descriptions de WSDL, dans le but de localiser des services. Nous présentons notre hypothèse et notre algorithme dans ce qui suit.

5.1. Hypothèse et algorithme

5.1.1. Hypothèse

Notre but est de faire de la composition de services web spécifiés sous forme de fichiers WSDL. À partir d'un ensemble d'inputs prédéterminés, nous désirons atteindre un ensemble de sorties. Les Figure 5.1 et Figure 5.2 illustrent notre but ainsi que les solutions possibles à notre problème.

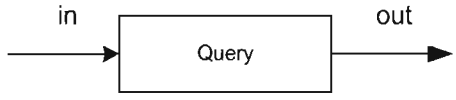


Figure 5.1. La requête (Query) (Mili et al., 2004)

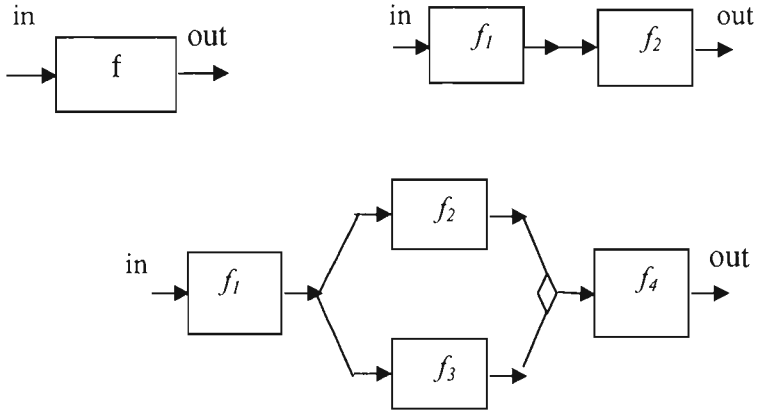


Figure 5.2. Quelques solutions possibles (Mili et al., 2004)

WSDL sépare la fonctionnalité abstraite d'un service web de sa mise en œuvre concrète. La fonctionnalité abstraite est décrite par des interfaces, appelées *portTypes* dans WSDL 1.1, qui représentent des collections d'opérations. Une opération, comme on l'a déjà mentionné, est décrite par un nom, une signature et une modalité d'invocation (*invocation modality*) – un sens/asynchrone ou double-sens/synchrone. La signature est une séquence de

messages d'entrée et de sortie générés selon des *call-patterns (In-Only, Robust In-Only, In-Out, In-Optional-Out, Out-Only, Robust Out-Only, Out-In, Out-Optional-In) (Mili et al., 2004)*. C'est grâce aux signatures des opérations, notamment aux messages d'entrées et de sorties et le type de leur entrée et de leur sortie qu'on compte faire notre composition de services web.

Nous construisons notre mécanisme de requêtes sur le principe suivant : les requêtes sur des services web doivent être elles-mêmes formulées comme la partie abstraite d'une spécification WSDL, à savoir : les messages, une interface et des opérations. Par conséquent, chercher un service web consistera en une recherche par appariement de signatures, i.e. appariement entre les signatures des opérations de la requête et les signatures des opérations des services web trouvés et qui appartiennent à l'industrie qu'on recherche. De plus, au lieu de chercher uniquement un appariement entre de simples opérations et la signature d'une opération d'une requête précise, nous pourrons aussi chercher des compositions d'opérations qui prennent le même message d'entrée que la requête et produisent le même message de sortie.

Bien que WSDL soit sémantiquement faible, nous croyons que la combinaison des références indexées avec la description précise des types de messages et de patrons nous aidera à découvrir les services web souhaités. Avec l'aide des dites références indexées, que ce soit pour les entités d'affaires ou pour les services d'affaires, nous pourrons restreindre la recherche de services aux entités d'affaires qui font partie de la bonne industrie, au lieu de faire une recherche plus étendue qui inclurait même les services d'autres industries. Ensuite, en partant de ces services trouvés, le type de leur message, ainsi que de ses patrons (*patterns*), nous pourrons déterminer si les services trouvés sont appropriés ou pas.

Ainsi, si nous revenons à l'exemple du voyageur qui souhaite un *forfait-voyage*, une opération qui prend comme message d'entrée (*input message*) le quadruple *<id du client, destination, date1, date2>* et comme message de sortie (*output message*) une liste de quadruples de la forme *<destination, date1, date2, prix >*, est probablement un service pour réserver des chambres d'hôtels ou un billet d'avion, si nous restreignons la recherche au domaine « voyages ».

5.1.2. Algorithme

Cette partie du mémoire est tirée de Mili et al., (2004). Avant d'entamer les explications, nous souhaitons insérer certaines définitions et exemples, afin de faciliter la compréhension du lecteur.

5.1.2.1 Définitions et exemples

Définition d'une réalisation

Une réalisation est un chemin comprenant un ou plusieurs services qui mène d'un ensemble de types A à un ensemble de types B. Pour mieux expliquer ce concept, nous procédons par un exemple. Soient A, B et C trois types. Soient A_to_B, A_to_C et B_to_C trois fonctions offertes dans un service web. La fonction A_to_B prend en entrée une donnée de type A et produit en sortie une donnée de type B, la fonction A_to_C prend en entrée une donnée de type A et produit une sortie de type C et enfin, la fonction B_to_C prend en entrée une donnée de type B et produit en sortie une donnée de type C.

Exemple d'une réalisation simple

Si notre but est de trouver une réalisation prenant en entrée une donnée de type A et donnant en sortie une donnée de type **C**, nous aurons comme résultat la réalisation suivante : *A -> A_to_B -> B -> B_to_C -> **C***. En d'autres mots, pour réussir à avoir une fonction prenant en entrée une donnée de type A et donnant en sortie une donnée de type C, nous avons fait la composition des deux fonctions : celle prenant en entrée une donnée de type A et donnant en sortie une donnée de type B et celle prenant en entrée une donnée de type B et donnant en sortie une donnée de type **C**. La composition de ces deux fonctions nous aura permis d'atteindre le résultat escompté. En voici une illustration dans la Figure 5.3.

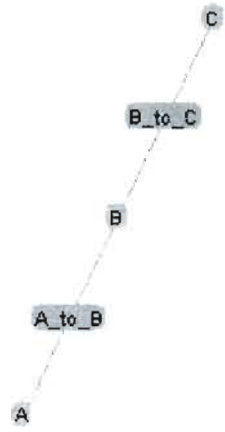


Figure 5.3. Une réalisation « simple »

Exemple d'une réalisation moins simple

Soient les opérations *A_to_B, A_to_C, A_to_D, B_to_C, B_to_D* et *C_to_D*. Nous nous donnons comme objectif d'avoir toutes les réalisations prenant en entrée une donnée de type A et donnant en sortie une donnée de type **D**.

Les réalisations trouvées dans ce cas sont les suivantes.

- *A -> A_to_D ->* ***D*** (Seule réalisation trouvée sans avoir recours à la composition de services)
- *A -> A_to_B -> B -> B_to_D ->* ***D***
- *A -> A_to_B -> B -> B_to_C -> C -> C_to_D ->* ***D***
- *A -> A_to_C -> C -> C_to_D ->* ***D***

La figure suivante nous montre toutes les réalisations trouvées.

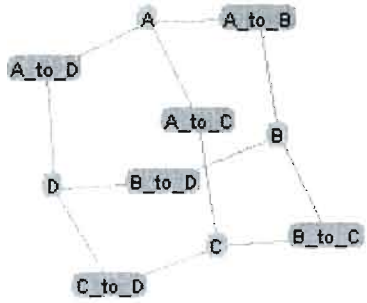


Figure 5.4. Une réalisation moins simple

5.1.2.2 Explications

Notre problème se résume essentiellement en une énumération de toutes les réalisations possibles d'une fonction g.

Nous pouvons aisément deviner que plusieurs réalisations peuvent être *équivalentes*.

Mais avant de comparer deux réalisations, il serait convenable de s'assurer qu'elles soient dans une forme normalisée : s'assurer qu'une fonction est invoquée dans notre réalisation le plus tôt possible, c.-à-d. aussitôt que tous ses inputs sont disponibles. Nous formalisons cette définition ainsi :

Soit $T_o(R)$ l'input de g, et $T_k(R)$ = Input(g) U ($U_{p=1 \text{ to } k}$ Output(F_p)), pour k = 1,..., s. $T_k(R)$ est, alors, l'ensemble des inputs qui peuvent être utilisés à l'étape (k+1). Une réalisation R est dite de *forme normalisée* si toute fonction f de R est invoquée à l'étape k tel qu'input(f) est un sous-ensemble de T_{k-1}, mais pas un sous-ensemble de T_{k-2} (T_{-1} étant l'ensemble vide).

Une fois cette étape terminée, nous nous intéressons à savoir si une réalisation est minimale ou pas. Pour ce, nous définissons le concept de couverture minimale (*minimal cover*) : Soit F une collection de fonctions dont l'ensemble de sorties contient un ensemble U. On dit que F est une couverture de U. F sera une *couverture minimale* de U si pour tout f de

F, F- { f } ne couvre plus U. Nous sommes maintenant prêts à déterminer si une réalisation R est une réalisation minimale : Soit R une réalisation de forme normalisée, composée des fonctions F_k, k variant de 1 à s, d'une fonction g.

R est une réalisation minimale de g si et seulement si F_k est une couverture minimale de (Input(F_{k+1}) U Output(g) INTER Output(F_k)) – T_{k-1}(R) pour k allant de 1 à s-1 et F_s est une couverture minimale de (Output(g) INTER Output(F_s)) – T_{s-1}(R). En d'autres termes, R est une réalisation minimale si toute fonction f invoquée à une étape k est nécessaire pour produire ou bien un nouveau type d'Output(g) ou bien un nouvel input pour la prochaine étape. Toute cette démarche est résumée dans l'algorithme qui suit (Figure 5.5), qui est le cœur de ce projet de recherche.

5.1.2.3 Algorithme EnumerateRealizations

```
Algorithm EnumerateRealizations (     Input k :StageNumber;
                                      Input Fk-1 :CollectionOfFunctions;
                                      Input Tk-2 :SetOfTypes;
                                      Input g : AFunction;
                                      Output all the realizations with prefix {F1,
                                             F2,..., Fk-1})
begin

(1) Tk-1 ← Tk-2 ∪Output (Fk-1);
(2) Ck-1 ← {f | Input (f) is a subset of Tk-1 but not of Tk-2};
(3) for every subset H of Ck-1 do
        (4) if Fk-1 is a minimal cover of (Input (H) ∪Output (g) ∩Output (Fk-1))-
                                                                        Tk-2
          then
                (5) if Tk-1 ∪Output (H) contains Output (g) then
                        (6) if H is a minimal cover of Output (g) ∩Output(H) -
                                Tk-1 then
                                (7) display the realization
                        end
                        (8) elsif Output (H) is not contained in Tk-1 then
                                (10) Fk ← H;
                                (11) EnumerateRealizations (k+1, Fk, Tk-1, g)
                        (12) end
                (13) end
        (14) end
```

```
end EnumerateRealizations
```

Figure 5.5. Algorithme EnumerateRealizations (Mili et al.)

5.2. Algorithme de génération de composition à partir de réalisations

5.2.1. Explications

Une fois une réalisation trouvée, il s'agit de générer toutes les compositions possibles de cette même réalisation. Chaque composition sera une énumération des étapes qu'on a accomplies avant d'arriver à la donnée désirée.

Quand une réalisation est *simple*, cette génération est aussi *simple*. Prenons l'exemple illustré dans la Figure 5.3, où la donnée souhaitée est de type C. Voici les étapes franchies pour atteindre une donnée de type C (Figure 5.6) :

À la phase 0, le type A utilisé par la fonction A_to_B provient de la fonction fictive f_0, et à la phase 1, le type B utilisé par la fonction B_to_C provient de la fonction A_to_B. Et c'est ainsi que le type C trouvé provient de B_to_C. En somme, nous avons consommé les types A et B, en passant par les fonctions A_to_B et B_to_C pour atteindre une donnée de type C.

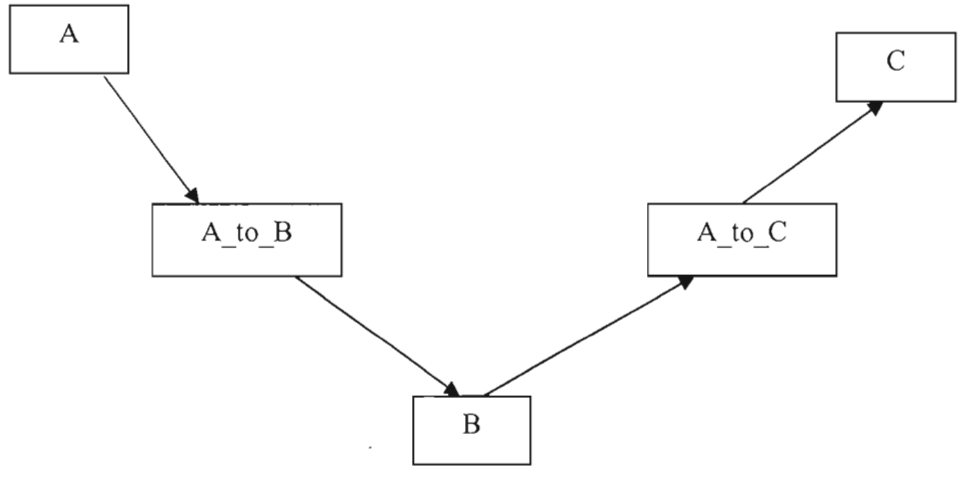


Figure 5.6. Composition générée à partir de la réalisation illustrée dans la Figure 5.3

Les réalisations ne sont pas toujours *simples*. En particulier, pour certaines réalisations plus complexes, il est important de noter que certains nœuds représentants des types, doivent être dupliqués car plusieurs fonctions/opérations ne peuvent utiliser la même instance d'un type. Il faut donc transformer les références à un type donné en des références à des variables de ce type. Pour illustrer ceci, nous présentons les Figure 5.7 et Figure 5.8 :

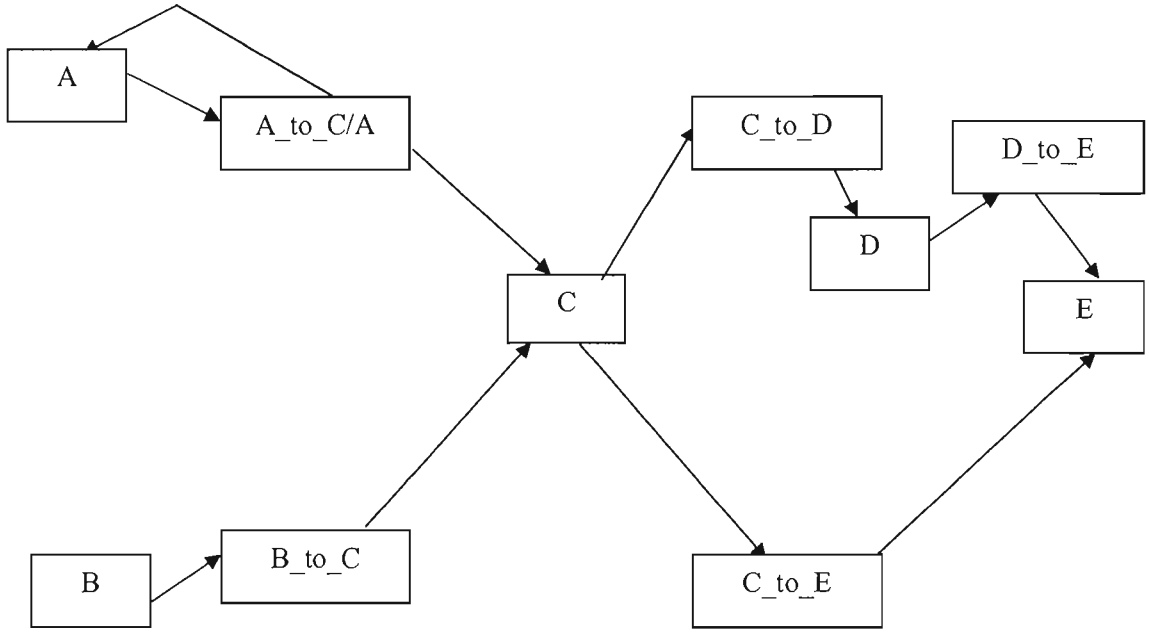


Figure 5.7. Réalisation avant de transformer les références aux types en références à des instances du même type

Explications :

- Il y a deux explications au fait que l'opération A_to_C/A prend une variable de type A et retourne une variable de type A comme sortie : A peut être la même variable, passée en entrée/sortie en paramètre, ou ce sont deux variables différentes du même type A.

- Deux opérations ne peuvent donner la même donnée comme sortie. Dans la Figure 5.7, les opérations A_to_C et B_to_C semblent donner la même donnée C comme sortie. Dans la réalité, ils donnent comme sorties deux données différentes, mais du même type C (Figure 5.8).

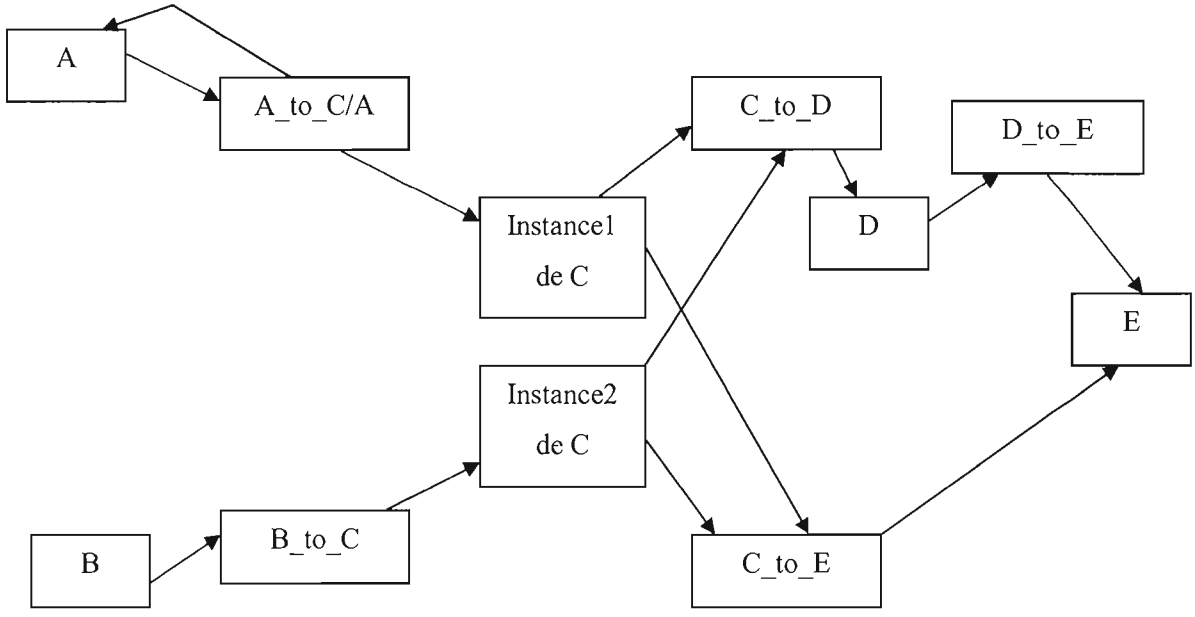


Figure 5.8. Réalisation où les références sont aux instances du type (flux de données)

Dans ce cas, les mises en oeuvres possibles des réalisations trouvées dans la Figure 5.7 sont les suivantes.

- *A -> A_to_C -> Instance1 de C -> C_to_E -> E*
- *A -> A_to_C -> Instance2 de C -> C_to_E -> E*
- *A -> B_to_C -> Instance1 de C -> C_to_D -> D -> D_to_E -> E*
- *A -> B_to_C -> Instance2 de C -> C_to_D -> D -> D_to_E -> E*

5.2.2. Algorithmes

La professeure Odile Marcotte a proposé les algorithmes suivants pour générer un flux de données à partir d'une réalisation.

Algorithme EnsDeFonction :T représente l'ensemble des types $\{t_1, t_2, \ldots, t_n\}$. Les fonctions disponibles sont $\{f_1, f_2, \ldots, f_m\}$. La fonction f_0 est une fonction fictive produisant les entrées de g.

EnsDeFonction(k,t) est l'ensemble des fonctions qui ont produit le type t avant le début de la phase k. Par exemple, si t est une des entrées de g, f_0 appartient à EnsDeFonction(k,t) pour tout k.

La Figure 5.9 présente bien cet algorithme.

```
Procedure EnsDeFonction ()
Début
      Pour chaque t appartenant à Input(g) Faire
            EnsDeFonction(1,t) := {f_0}
      Fin Pour
      Pour k :=2 HAUT s Faire
            Pour chaque f appartenant à F_{K-1} Faire
                  EnsDeFonction(k,t) := EnsDeFonction (K-
                                                      1, t)
                  Pour chaque t appartenant à Output(f) Faire
                        Insérer f dans EnsDeFonction(k,t)
                  Fin Pour
            Fin Pour
      Fin Pour
Fin EnsDeFonction
```

Figure 5.9. Procédure EnsDeFonction

La Figure 5.10 nous présente le résultat de la procédure `EnsDeFonction` appliquée à l'exemple de la Figure 5.7 précédente.

	A	*B*	*C*	*D*	*E*
k=1	f_0	f_0	Ø	Ø	Ø
k=2	f_0, *A_to_C/A*	f_0	*A_to_C/A*, *B_to_C*	*C_to_D*	Ø

Figure 5.10. Résultat de la procédure `EnsDeFonction` *sur l'exemple de la Figure 5.7*

Algorithme_Suite : Cet algorithme crée une collection de triplets (k, f, t), où k représente une phase, f une fonction de F_k et t un type de input(f). Pour énumérer toutes les mises en oeuvre de la réalisation de g, il est indispensable de considérer chaque entrée de la fonction f et de choisir, parmi toutes les fonctions ayant produit cette entrée, une seule fonction.

L'algorithme *Suite* est présenté dans la Figure 5.11.

```
Procédure Suite()
Début
        Suite := suitevide
        Pour K :=1 HAUT s Faire
                Pour chaque f appartenant à F_K Faire
                        Pour chaque t appartenant à Input(f) Faire
                                Ajouter le triplet (k,f,t) à la fin
                                                        de suite
```

```
                    Fin Pour
                Fin Pour
            Fin Pour
Fin Suite
```

Figure 5.11. Procédure Suite

Algorithme ÉnumérerImplatations : Cet algorithme est récursif. Longueur représente la longueur de la collection Suite, r est la position du triplet considéré dans Suite. Quant au tableau Tab, il se définit comme suit : si r est la position du triplet (k, f, t) dans la collection Suite, Tab[r] contient la fonction choisie pour produire t, lorsque t est utilisé à la phase k en tant qu'entrée de f . La Figure 5.12 présente cet algorithme.

```
Procedure ÉnumérerImplantations(r : Entier positif,
                                E/S Tab : TypeTableau)
Début
    Si r<= Longueur Alors
        (k,f,t):= suite[r]
        Pour chaque f appartenant à EnsDeFonction(k,t)Faire
            Tab[r]:= f
            ÉnumérerImplantations(r+1, Tab)
        Fin Pour
    Sinon
        AfficherImplantations(Longueur, Suite, Tab)
    Fin Si
Fin ÉnumérerImplantations
```

Figure 5.12. Procédure ÉnumérerImplantations

Pour afficher toutes les mises en oeuvre possibles, il suffit de faire l'appel ÉnumérerImplantations(1, Tab).

CHAPITRE 6

MISE EN oeuvre ET TESTS

6.1. Mise en oeuvre

6.1.1. Flux de traitement

Nous donnons ici une idée générale du fonctionnement de notre programme.

Au départ, nous avons besoin de trois entrées.

- Une collection de fichiers WSDL à lire.
- Un fichier WSDL (nommé Target_X), où sont spécifiées les données, c'est-à-dire:
    - L'input qu'on détient au début (input de départ).
    - La sortie qu'on désire obtenir.
- Un fichier texte (nommé propertyFile) où sont spécifiés :
    - Le chemin pour lire la collection de fichiers WSDL.
    - Le chemin pour lire le fichier de données.

Nous chargeons les données lues en mémoire et nous lançons le programme. Le premier résultat, une réalisation (Figure 5.7), est affiché dans une fenêtre extérieure d'*Eclipse*

en forme de graphe obtenu grâce à la bibliothèque *Prefuse*. Ensuite, on passe l'objet réalisation trouvé aux algorithmes énumérant des mises en oeuvre (section 5.2.2), et on affiche les résultats dans une fenêtre éclipse sous forme de texte.

La figure suivante nous montre ce déroulement.

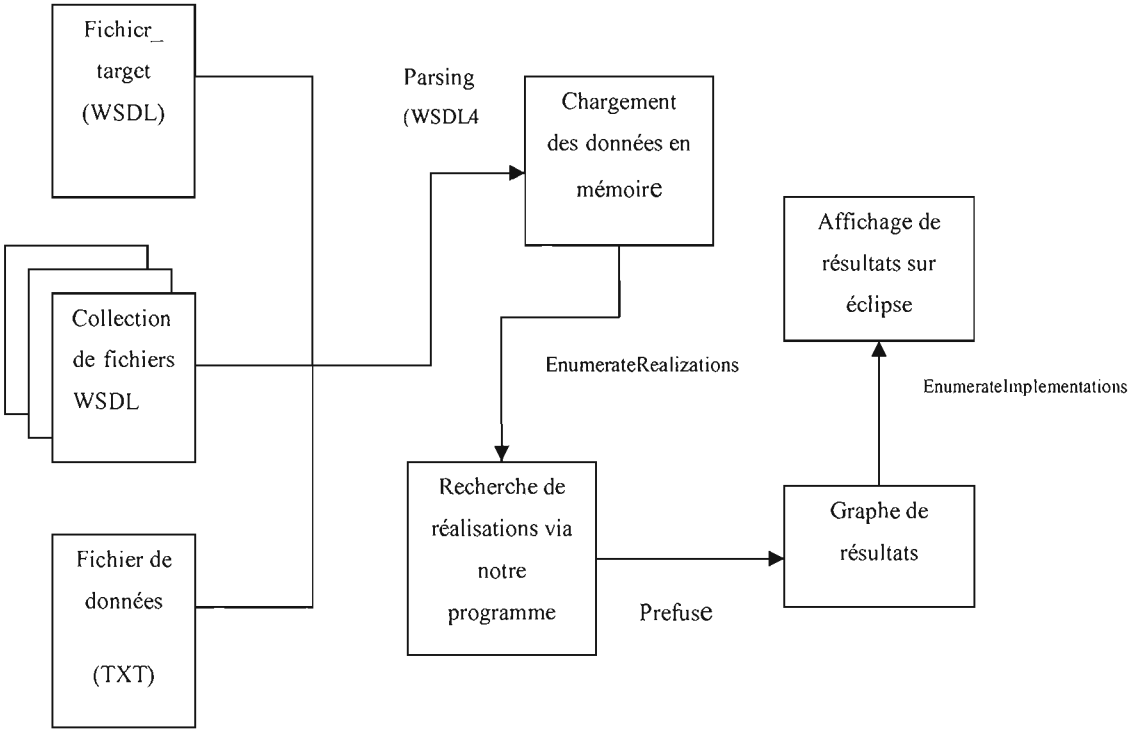


Figure 6.1. Flux de traitement de notre programme

6.1.2. Mise en oeuvre des algorithmes

6.1.2.1 Bibliothèques utilisées

Avant d'aborder notre approche pour mettre en oeuvre nos algorithmes, il convient de présenter les bibliothèques utilisées à cette fin.

6.1.2.2 WSDL4J

Introduction

Nous avons utilisé WSDL4J pour beaucoup de raisons. D'abord, c'est une bibliothèque ouverte (*open source*) développée par IBM en langage Java. Ensuite, elle permet l'accès aux aux éléments d'un document WSDL par catégorie ou par nom, ce qui est assez convenable et pratique pour notre application.

Par ailleurs, WSDL4J permet la création, la représentation et la manipulation des documents WSDL qui décrivent des web services (WSDL4J). Cet outil, au lieu de *parser* manuellement les documents et accéder à leurs informations en traversant l'arbre créé par le parser, accède directement aux méthodes et aux paramètres et retourne toutes les valeurs et les informations contenues dans un document WSDL.

Installation

D'abord, il faut télécharger le package se trouvant à l'Url suivante : *http://www-124.ibm.com/developerworks/projects/wsdl4j/* et installer les archives disponibles sur l'ordinateur. Ces archives contiennent dans le répertoire lib deux jars : wsld4j.jar et qname.jar, qu'il faut rajouter au *classpath* (WSDL4J).

Exemple d'utilisation de WSDL4J

Dans l'exemple suivant, la bibliothèque WSDL4J est utilisée pour obtenir une liste de toutes les méthodes contenues dans le document WSDL du service web.

```
1 private void getMethods() {
2     try {
3         WSDLFactory wsdlfactory = WSDLFactory.newInstance();
4         WSDLReader reader = wsdlfactory.newWSDLReader();
5         Definition def = reader.readWSDL(endpointUrlString +
          "?WSDL");
6         Map ports = def.getPortTypes();
7         Iterator it = ports.values().iterator();
```

```
 8          PortType type = null;
 9          while (it.hasNext()) {
10              PortType typeTmp = (PortType) it.next();
11              QName pn = typeTmp.getQName();
12              if (pn.getLocalPart().equals(SERVICE_NAME)) {
13                  type = typeTmp;
14              }
15          }
16          java.util.List methodsList = type.getOperations();
17          Iterator itMethods = methodsList.iterator();
18          while (itMethods.hasNext()) {
19              Operation op = (Operation) itMethods.next();
20              methods.add(op.getName());
21          }
22      } catch (Exception e) {
23          System.out.println("exception in getmethods: " +
            e.getMessage());
24      }
25  }
```

Figure 6.2. Exemple d'utilisation de WSDL4J

6.1.2.3 Prefuse

Le contexte

Pour afficher nos résultats et les rendre plus lisibles à l'œil nu, nous avons utilisé la bibliothèque *Prefuse* qui est disponible sur le web. Grâce à cette bibliothèque, nous avons pu afficher nos résultats en forme de graphe comportant plusieurs nœuds. Les nœuds se partagent en deux familles : des nœuds qui représentent des types et des nœuds qui représentent des opérations. Si une opération prend un type en entrée, alors le nœud qui représente le type en question pointe vers un autre nœud qui, lui, représente l'opération en question.

Introduction à l'outil Prefuse

Prefuse est un cadre d'application (*framework*) utilisé pour aider les développeurs à créer une application visuelle en utilisant le langage Java. Ce *framework* peut être utilisé pour construire des applications autonomes ou des composantes qui font partie d'une plus grande application, ou encore comme *web applets*. La préoccupation principale de *Prefuse* est de simplifier le processus de représentation des données et leur visualisation. Pour ce faire, *Prefuse* utilise des dispositifs tels que (Prefuse).

- Structures de tables, de graphes et d'arbres permettant de représenter, d'indexer et d'interroger des données de formats divers, tout en offrant une gestion efficace de la mémoire.

- Composants d'affichage, de couleur, de taille, des techniques de distorsion, d'animation, etc.

- Une bibliothèque de commandes d'interaction pour les opérations ordinaires d'interaction ou de manipulations directes.

- Un API simple.

- Une technique d'animation fondée sur un mécanisme de planification d'activités.

- Des requêtes de type dynamique pour un filtrage interactif de données.

- Recherche de textes à l'aide de moteurs de recherche déjà existants.

- Un moteur de simulation pour affichages et animations.

- Un support de requêtes SQL pour pouvoir interroger des bases de données et transformer les résultats en structures de données de la bibliothèque elle-même.

La structure de Prefuse

L'architecture de l'outil *Prefuse* est fondée sur un modèle d'architecture qui divise le processus de visualisation en plusieurs étapes (Chi, 2000), allant de l'acquisition et la modélisation des données, au codage de leur visualisation et à leur affichage interactif.

La figure suivante résume cette démarche.

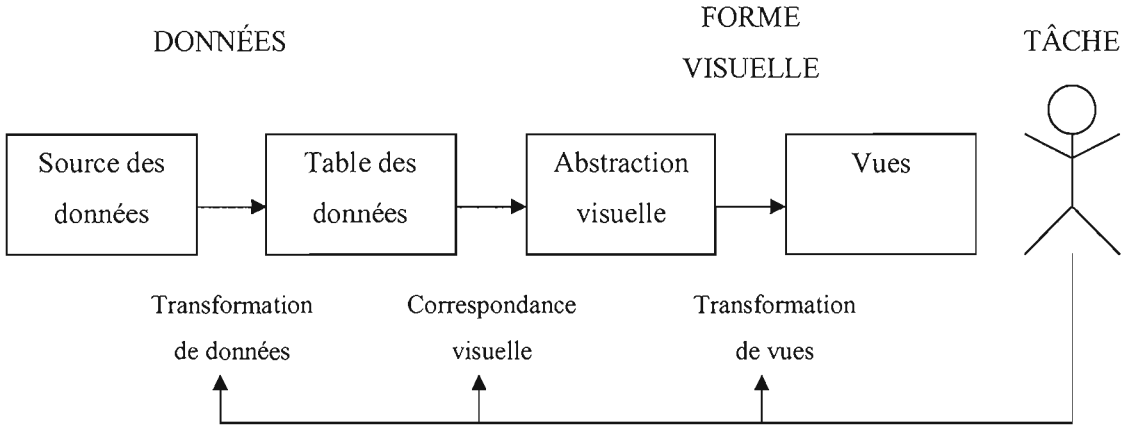


Figure 6.3. The prefuse visualization framework (inspirée de : Presuse, Toolkit Structure)

Les données reçues sont transformées en tables qui nous permettent de construire une *abstraction visuelle* : on modélise les propriétés visuelles telles que la couleur, la position ou la forme géométrique. Ensuite, à partir de cette abstraction visuelle, des vues interactives des données sont créées, avec potentiellement l'interaction de l'utilisateur qui pourrait apporter des changements à n'importe quelle étape du *framework.*

Construction d'une application à l'aide de Prefuse

Une application est bâtie, à l'aide de l'outil *Prefuse*, en suivant les étapes suivantes (Prefuse) :

- Charger les données et les stocker dans une structure de données de notre choix.

- Créer une visualisation qui transforme ces données en visualisation abstraite. Tables, graphes et arbres sont ensuite rajoutées à la visualisation et référencés par un nom unique.

- Créer une *RendererFactory* et la mémoriser avec la visualisation. Cette classe de « fabrication » se charge d'attribuer des *Renderers* (Module de programme utilisé pour produire le rendu d'une image de synthèse) aux items visuels.

- Construire une série d'actions qui s'occupent de l'abstraction visuelle. Ces actions fixeraient, par exemple, les attributs visuels des items, comme leur couleur, leur taille et leur format. Les instances de ces actions se regroupent dans une *ActionList* pour pouvoir traiter différentes données.

- Initialiser un ou plusieurs affichages interactifs, afin de manipuler et de visualiser les items. Spécifier les comportements interactifs en rajoutant des contrôles (*Controls*) aux affichages. La recherche et le filtrage de données est possible grâce au rajout d'un système de requêtes dynamique.

6.1.2.4 Mise en oeuvre de l'algorithme *EnumerateRealizations*

L'article sur lequel nous nous sommes basés (Mili et al., 2004) présentait l'algorithme *EnumerateRealizations (Cf. annexes)* avec les paramètres d'entrée suivants : k (*stageNumber*), F_{k-1} (*Collection of functions*), T_{k-2} (*set of types*) et g (*function*). Nous avons jugé nécessaire de rajouter les paramètres d'entrée suivants.

- *AvailableOperations* : Une collection de toutes les opérations disponibles. Nous chargeons cette information au début lors de la lecture de tous les fichiers WSDL auxquels nous avons accès. La nécessité de cette information nous paraît évidente : il faut avoir accès à ces opérations durant la composition de services web pour faire une recherche exhaustive.

- *Realization* : L'algorithme *EnumerateRealizations* est récursif. Au fur et à mesure que les appels récursifs sont faits, il est nécessaire de garder en mémoire la réalisation (*realization*) partielle que nous avons construite au stade k de la recherche.

- *PartParents* : Une collection qui représente les dernières opérations rajoutées à l'arborescence de la réalisation. Nous détaillerons dans une sous-section les structures de données utilisées pour représenter les entrées et les sorties, mais pour l'instant, il convient d'indiquer que ce choix a été fait pour garder le lien entre les nœuds représentant les opérations et ceux représentant les *parts* (types).

Ce sont les seuls changements que nous avons jugés pertinents à l'algorithme *EnumerateRealizations* : le reste a été codé tel quel.

6.1.2.5 Les autres Algorithmes

Il n'y a eu aucun changement apporté aux algorithmes générant un programme exécutable à partir d'une réalisation. Ils ont été codés tels quels.

6.1.3. Structures de données

Avant de présenter les structures de données utilisées pour représenter les données importantes de notre programme, nous donnons une idée de la manière dont on a construit la solution de notre problème.

À partir de notre entrée de départ, nous avons construit un nœud. Ce nœud, de type *RealizationNodePart*, est le seul à ne pas avoir d'antécédents. Les opérations qu'on peut atteindre à partir de cette entrée, seront les enfants de ce nœud. Chaque collection d'opérations aura un nœud la représentant à son tour. Ce nœud, de type *RealizationNodeOperation*, aura comme parents les nœuds représentants les *parts* qui ont permis de l'atteindre (entrées), et comme enfants, les parts que ses opérations donnent comme sorties. Et ainsi de suite, jusqu'à ce qu'on trouve la collection de parts qu'on désire atteindre.

La Figure 6.4 donne un aperçu de la structure de notre solution.

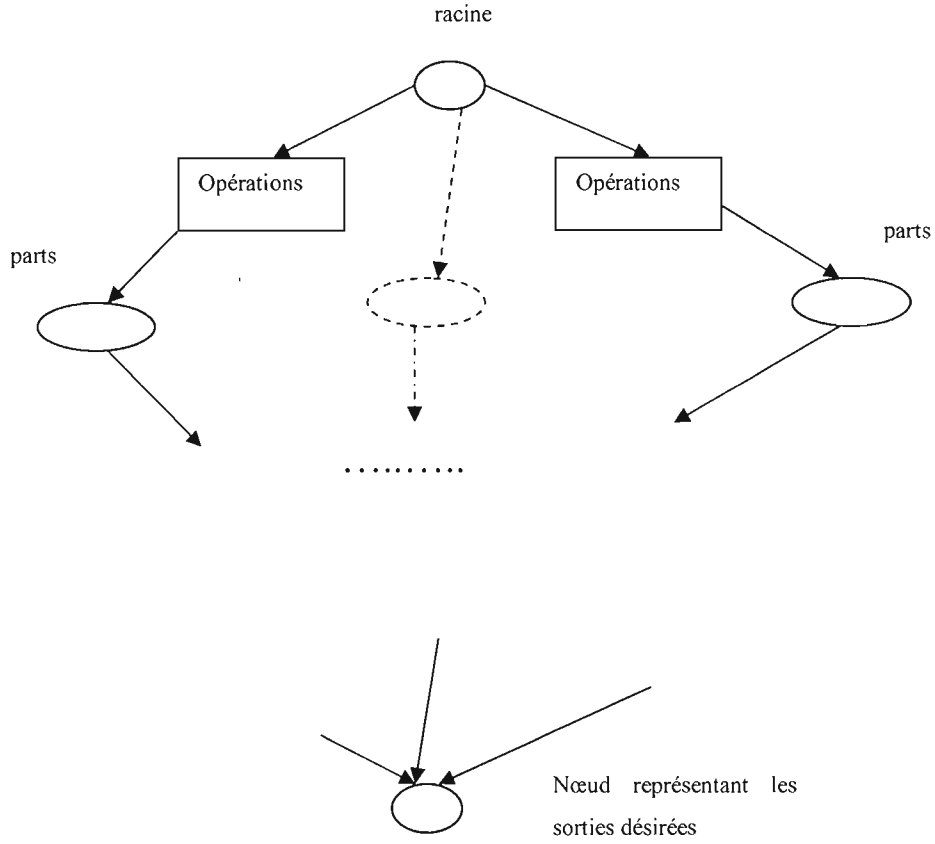


Figure 6.4. Structure de la première étape de notre solution

Nous présentons maintenant les structures de données de nos principales entrées et sorties :

- *Realization* : Une instance de cette classe est composée d'un ensemble de nœuds. Un nœud peut être de type *RealizationNodePart* ou *RealizationNodeOperation*.

- *RealizationNode* : Est une classe abstraite représentant un nœud.

- *RealizationNodePart* : Est une spécialisation de la classe *RealizationNode*. Une instance de cette classe possède les attributs *parts*, *partParent* et

partChildren. Ces attributs sont des collections. La collection *parts* représente tous les *parts* inclus dans ce nœud. *partParent* est la collection des parents de ce noeud et *partChildren* la collection de ses enfants.

- *RealizationNodeOperation* : Est une spécialisation de la classe *RealizationNode*. Une instance de cette classe possède les attributs suivants.
  - Une opération : L'opération que ce nœud représente.
  - *stageNumber* : À quel stade de la réalisation l'attribut opération a été rajouté.
  - Une collection des parents de l'attribut *operations* : Ce sont les *parts* qui ont permis d'atteindre cette opération.
  - Une collection des enfants de l'attribut *operations* : Ce sont les *parts* que produit cette opération (sorties).

Avant de décrire brièvement les classes principales de notre programme, nous présentons un diagramme de classe.

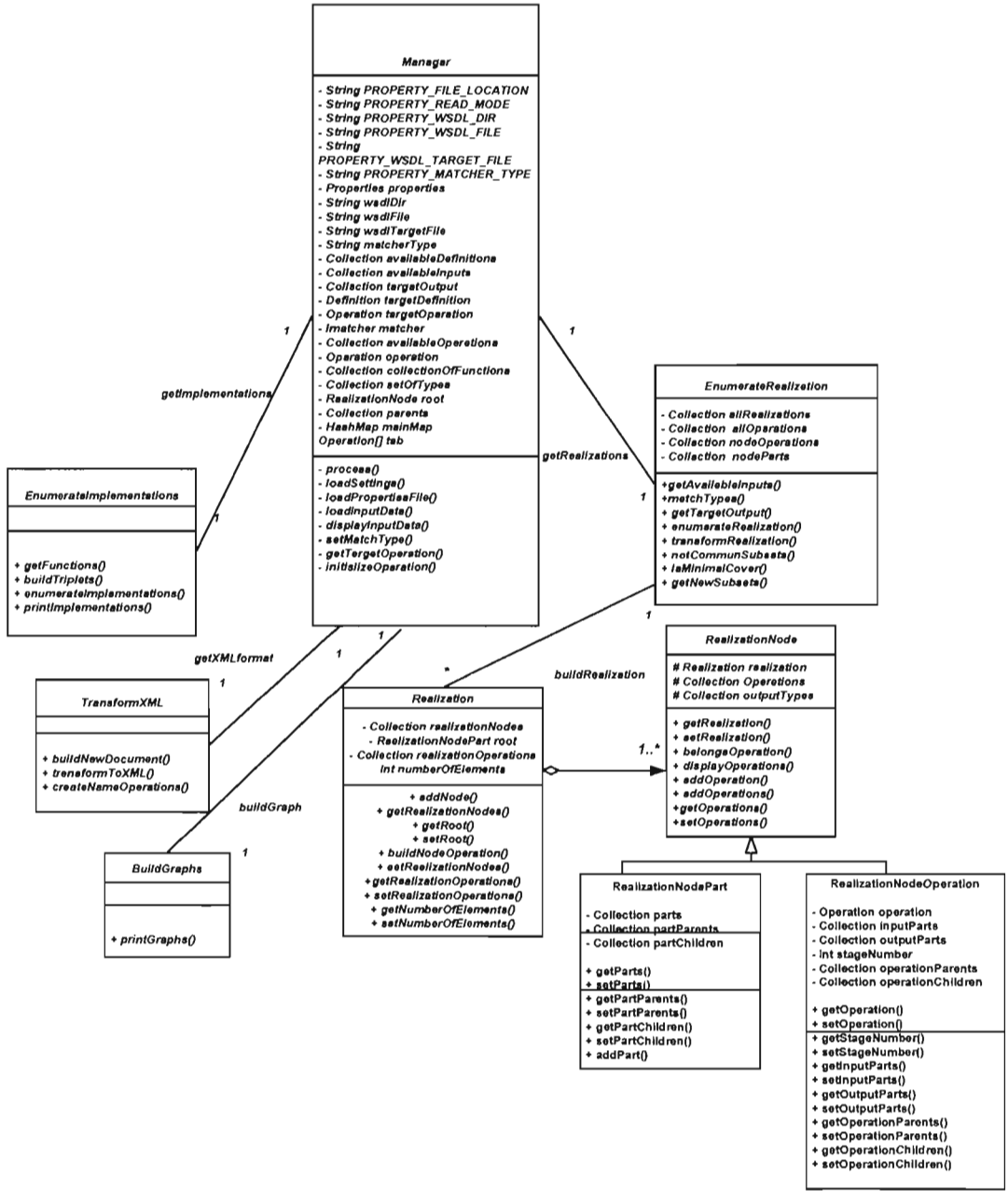


Figure 6.5. Diagramme de classe de notre programme

- La classe *Manager* : Classe contenant le programme principal (*main*) de notre application. C'est à partir de cette classe qu'on lit nos fichiers, qu'on les charge en mémoire, qu'on définit nos attributs et qu'on fait appel à nos

algorithmes *EnumerateRealizations* et *EnumerateImplementations,* puis qu'on lance la construction du graphe illustrant la solution trouvée.

- La classe *EnumerateRealization* : Classe contenant des attributs et des méthodes statiques : Il n'y a pas d'objet *EnumerateRealization.* C'est une classe qui contient la mise en oeuvre de l'algorithme *EnumerateRealizations* et une partie de la logique d'affaires.

- La classe *EnumerateImplementations* : Classe contenant des méthodes statiques. Elle contient tous les algorithmes prenant en charge de transformer la réalisation trouvée à la première étape en une réalisation « *réaliste* ».

- La classe *TransformXML* : C'est cette classe qui contient les méthodes nécessaires à la transformation de notre première réalisation en un document DOM, avant de l'envoyer à *BuildGraphs* et d'en faire le graphe. Cette classe contient des méthodes statiques aussi. Elle ne représente aucun objet.

- La classe *BuildGraphs* : Classe contenant la méthode statique *printGraph*, qui dessine le graphe et l'affiche grâce à *Prefuse*.

6.1.4. Format des entrées

En ce qui concerne l'analyse des données du fichier en vue d'une composition de services fondée sur l'appariement des types, notre programme ne supporte pas certaines formalités de WSDL.

- *Import* : Les fichiers WSDL qui contiennent des imports dont on a besoin dans l'analyse des données.

- *Restriction* : Les types complexes qui sont des restrictions d'autres types.

L'analyse de ce genre de fichiers ne présente pas de difficultés supplémentaires. Par manque de temps, nous avons tout simplement préféré nous concentrer sur des fichiers WSDL *de base* pour valider nos algorithmes. Ainsi, nous étions sûrs que si un test ne marchait pas, ce n'était pas à cause des données lues.

6.1.5. Format de la sortie

L'algorithme *EnumerateRealizations* retourne un objet de type Realization. Nous appelons alors la méthode *transformRealization()* qui prend les nœuds de cette opération et les transforme en nœuds d'un document DOM. Une fois le document DOM construit, nous l'envoyons à la méthode *printGraph()*, qui utilise *Prefuse* pour afficher notre réalisation. À cette étape, la réalisation affichée n'est toujours pas « *réaliste* » (se référer à la section 5.2.1). Nous appliquons ensuite les algorithmes d'énumération des réalisations (section.5.2.2) se référant à des instances du type et nous affichons la solution en format texte.

6.1.6. Algorithme d'appariement utilisé pour les tests

L'algorithme utilisé pour les teste est basé sur l'équivalence des noms (*name equivalence*). Il considère les types comme des arbres, dont les nœuds possèdent un ensemble d'enfants non-ordonnés. Nous énonçons ici deux concepts importants qui déterminent si deux nœuds sont équivalents (Mili et al., 2004).

- Deux nœuds qui ont le même espace de nom sont considérés équivalents.
- Deux nœuds qui ont le même nom, mais un espace de nom différent sont équivalents, si et seulement si leurs enfants, i.e. leur arborescence sont équivalents un à un.

Il est à noter que dans un contexte de schémas XML, pour vérifier l'équivalence de deux éléments, il ne suffit pas de vérifier l'équivalence de leurs types, mais aussi de vérifier qu'ils ont la même multiplicité : Nous n'avons pas besoin de nous assurer qu'ils ont la même cardinalité. Il suffit de faire la distinction entre un élément qui n'apparaît qu'une seule fois, et un élément qui a plusieurs occurrences.

L'algorithme basé sur la stratégie de *name equivalence* est décrit ici.

```
Algorithm NameEquivalence(Input T1 : Type ;Input T2 : Type;
                                Output Answer : boolean);
begin
        if (T1.name = T2.name) then
                if (T1.name_space = T2.name_space) then
                        Answer <- true;
                else
                        SortedChildrenT1 <- children of T2 sorted by name
                        SortedChildrenT2 <- children of T2 sorted by name
                        Done   <- false
                        Answer <- true
                        while (not done)
                                Answer <- Answer ^ NameEquivalence(
                                                  SortedChildrenT2.next);
                                done      <-      ( (SortedChildrenT1 empty)
                                                          v
                                                  (SortedChildrenT1 empty)
                                                          v
                                                  answer = false)
                        endwhile
                endif
        else
                Answer <- false
        endif
end NameEquivalence
```

Figure 6.6. Algorithme basé sur l'équivalence des noms

6.1.7. Fichiers testés

La première partie des tests a été faite sur des fichiers locaux que nous avons élaborés nous-mêmes. Dans la Figure 6.7, nous présentons un exemple des fichiers lus et testés par notre programme.

```
<?xml version="1.0"?>
<definitions name="linear_single_service"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

```
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   targetNamespace="urn:IRSACatalogSearch"
   xmlns:tns="urn:IRSACatalogSearch">


        <message name="A"><part name="A" type="xsd:A"/></message>
        <message name="B"><part name="Output"
                                        type="xsd:B"/></message>

        <portType name="SamplePortType">
                <operation name="A_to_B"><input
               message="tns:A"/><output message="tns:B"/></operation>
        </portType>
        <binding name="SampleBinding" type="tns:SamplePortType">
                <http:binding verb="GET"/>
                <operation name="A_to_B"><http:operation
location="cgi-bin/Oasis/CatList/nph-
catlist"/><input><http:urlEncoded/></input><output><mime:content
type="text/xml"/></output></operation>
        </binding>
        <service name="SampleService">
                <port name="SamplePort" binding="tns:SampleBinding">
                        <http:address
location="http://irsa.ipac.caltech.edu/"/>
                </port>
        </service>
</definitions>
```

Figure 6.7. Exemple de fichiers WSDL testés par notre programme

Dans ce fichier, nous avons un *part* de nom « A » et de type « A », ainsi qu'un *part* de nom « *output* » et de type « B ».

Dans un autre fichier WSDL appelé target_X, nous spécifions notre collection d'entrées disponibles au début de la recherche et ce que nous voulons comme résultat(s) (collection de sorties).

Ce fichier est décrit dans la figure suivante.

```
<?xml version="1.0"?>
<definitions name="linear"
targetNamespace="urn:xmltoday-delayed-quotes"
xmlns:tns="urn:xmltoday-delayed-quotes"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
        <message name="A"><part name = "input"
                                type="xsd:A"/></message>
      <message name="B"><part type="xsd:B"/></message>

        <portType name="TargetPortType">
                <operation name="A_to_B">
                        <input message="tns:A"/>
                        <output message="tns:B"/>
                </operation>
        </portType>
</definitions>
```

Figure 6.8. Exemple de fichier WSDL où sont décrits nos entrées et nos sorties

Dans cet exemple, nous spécifions que nous avons une entrée de nom « *input* » et de type « A » et que nous désirons atteindre un résultat de type « B ».

6.1.8. Affichage des réalisations et des compositions

Après lecture (analyse synthaxique) des fichiers donnés en exemple dans les deux figures précédentes, voici l'affichage que nous réussissons à avoir.

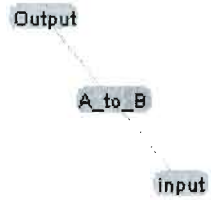


Figure 6.9. Résultat trouvé à la suite de l'analyse synthaxique des fichiers de la Figure 6.7 et la Figure 6.8

L'exemple donné illustre un cas très simple d'une réalisation qui, à partir d'une entrée de type « *input* », arrive à trouver une opération A_to_B qui donne comme sortie une instance d'un type « *output* ».

6.2. Les données de tests

6.2.1. Difficultés des tests

Après avoir codé l'algorithme et l'avoir testé sur des exemples construits par l'étudiant lui-même, il était temps de tester sa performance sur des tests réels. Par conséquent, il était primordial de trouver des fichiers WSDL sur le web pour faire nos tests. Nous avons cherché des fichiers WSDL publics et « réalistes » sur le web. Finalement, nous nous sommes basés sur les fichiers fournis par *Aleph Web Services Search Engine (Aleph)*, qui donne accès à plus de 8000 services web disponibles.

Format des fichiers :

Le test de ces fichiers ne fut pas de tout repos : la forme des fichiers était différente de celle des fichiers sur lesquels nous faisions nos tests auparavant. À titre d'exemple, la forme des messages varie d'un fichier WSDL à un autre.

Dans certains fichiers, un message s'écrit sous la forme.

```
<message name="echoResponse">
        <part element="tns:echoResponse" name="parameters"></part>
</message>
```

*Figure 6.10. Un part défini par un **element** et un **name***

Dans d'autres fichiers, il s'écrit sous la forme :

```
<message name="GreetingResponse">
<part name="return" type="xsd:string" />
</message>
```

*Figure 6.11. Un part défini par un **name** et un **type***

Ainsi, le premier message (*echoResponse*) contient un *part* défini par un élément (*element*) et un nom (*name*), alors que le deuxième (*GreetingResponse*) contient un *part* qui, lui, est défini par un nom (*name*) et un type. Ceci engendre quelques difficultés quand il est question de faire de l'analyse synthaxique et du *matching*.

Problèmes de l'analyse synthaxique:

L'autre problème auquel nous nous sommes butés est le peu de robustesse de WSDL4J. L'exemple le plus frappant fut celui-ci : à la lecture d'un fichier WSDL pris sur le web, une exception est levée. « `[Fatal Error] GRSearchService.wsdl:3:6: The processing instruction target matching "[xX][mM][lL]" is not allowed.` ». Nous avons cherché longtemps avant de comprendre que ce sont de simples sauts de ligne au début du fichier qui causaient cette exception.

Problèmes de performance (Java heap space) :

Dans la ligne de code suivante de l'algorithme *EnumerateRealizations* : `for every subset H of Ck-1 do,` nous explorons tous les sous-ensembles d'une collection d'opérations. Or, pour une collection de n opérations, l'ensemble de ces sous-ensembles aura une cardinalité de 2^n. Il est facile de déduire que plus le nombre d'opérations est élevé, plus le

nombre de sous-ensemble est (très) élevé. À un certain nombre d'opérations, l'exception suivante est levée : `Exception in thread "main" java.lang.OutOfMemoryError: Java heap space.`

Nous avons dû limiter le nombre de sous-ensembles afin d'éviter cette exception. Un nombre M de sous-ensembles est configuré dès le début et définit ce nombre-plafond de sous-ensembles explorés pour trouver la solution recherchée.

6.3. Les résultats

Les tests ont été faits en trois étapes. La première étape, comme nous l'avons mentionné auparavant, a été faite sur des fichiers construits par l'étudiant. Dans la deuxième étape, nous avons utilisé les fichiers WSDL du site Aleph pour faire nos tests. Dans la troisième étape, nous avons utilisé les fichiers WSDL du site *www.ws-challenge.org* pour nos tests. Nous expliquons dans ce qui suit la nécessité de la troisième étape et nous exposons nos résultats.

6.3.1. Première évaluation

Les fichiers construits localement avaient pour principal but de valider au fur et à mesure la mise en oeuvre des principaux algorithmes servant notre but de composition de services web par appariement des signatures. Néanmoins, nous savions que dès que la validation serait faite, elle ne serait considérée dans la validation de l'approche elle-même, car fondée sur des fichiers théoriques et construits localement.

6.3.1.1 Résultats

Les résultats de cette première évaluation furent tout à fait satisfaisants. Toutes les solutions espérées furent trouvées. Nous avons essayé les solutions faciles d'abord. Puis, graduellement, augmenté la difficulté des tests. Les solutions escomptées ont été trouvées à chaque test. Il convient de signaler que, comme la complexité de l'algorithme *EnumerateRealizations* est exponentielle, le programme prenait du temps à s'exécuter quand le nombre de solutions possibles était élevé.

6.3.2. Deuxième évaluation

Une fois la première évaluation faite, nous avons voulu tester notre approche sur des fichiers offerts sur le web. Nous avons collecté nos fichiers WSDL du site d'*Aleph* (mentionné ci-haut). Nous cherchions les fichiers par domaines, en introduisant un mot-clé dans la barre de recherche du site. « *wine* », « *travel* », « *sport* » ou encore « *boxe* » figurent parmi les mots ou domaines testés (Figure 6.12).

Figure 6.12. Moteur de recherche des services web du site Aleph

Le moteur de recherche du site Aleph nous affichait ensuite tous les services web reliés au mot-clé que nous avions soumis (Figure 6.13).

Figure 6.13. Extrait des fichiers proposés par le moteur de recherche des services web du site Aleph

Ensuite, il fallait sauvegarder chacun de ces fichiers, les lire un par un et charger leurs définitions en mémoire au fur et à mesure, avant d'effectuer des tests manuellement et d'imaginer des réalisations théoriquement possibles sur papier.

Le site d'*Aleph* donne certains détails sur chacun des fichiers WSDL qu'il propose. En particulier, il fournit le nom du fichier, son url, son cache en XML et en HTML, ainsi que le pays où il est localisé, et présente toutes les interfaces du WSDL (Figure 6.14).

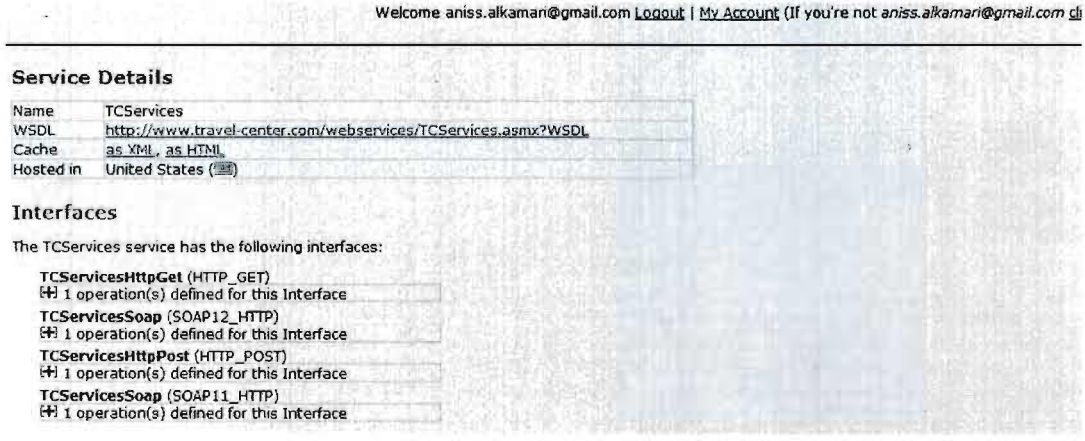


Figure 6.14. Détails sur un service web trouvé sur le site d'Aleph

Le site d'*Aleph* fournit aussi une idée sur la disponibilité du fichier (*response time measured*).

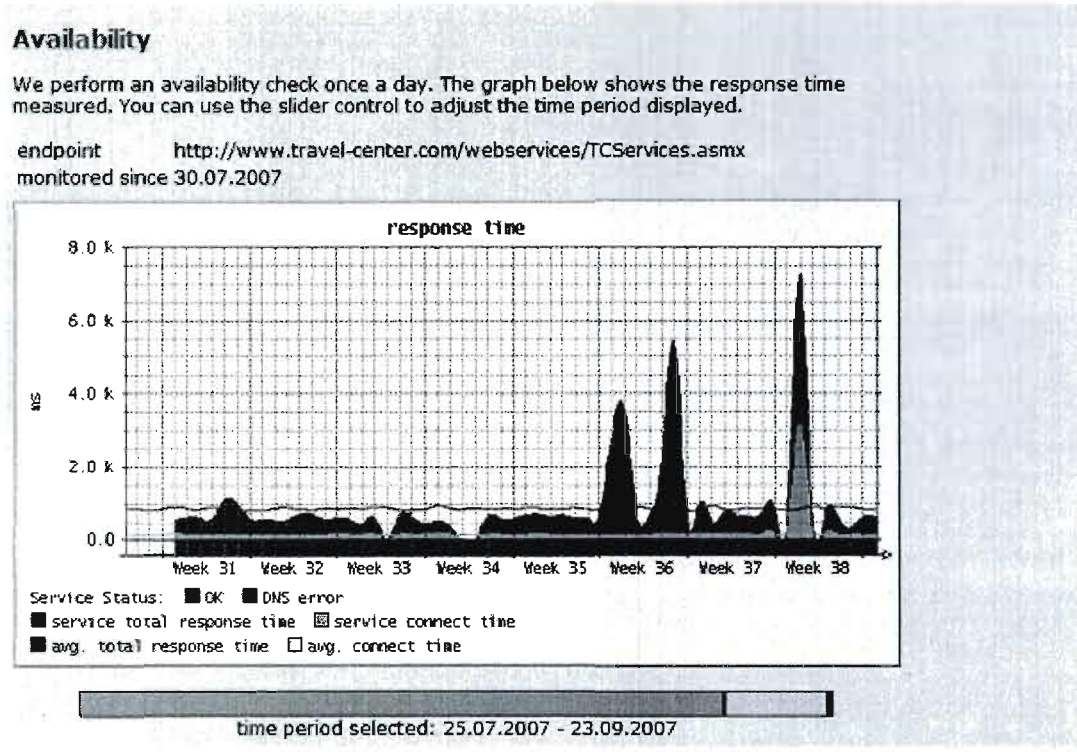


Figure 6.15. Disponibilité d'un service web du site Aleph

6.3.2.1 Résultats des tests de la deuxième évaluation

Pour illustrer les résultats de nos tests, nous choisissons un mot-clé, nous citons le nombre des fichiers que sa recherche nous a donnés sur le site d'Aleph, et nous donnons un exemple des fichiers trouvés.

Mot-clé recherché : « *travel* ».

Nombre de fichiers trouvés : 43.

Nous présentons dans ce qui suit trois exemples de fichiers trouvés, que nous décortiquons au fur et à mesure. Ensuite, nous faisons une analyse de ces exemples afin d'arriver à notre conclusion. Ce que nous voulons souligner par ces trois exemples, qui ne

sont qu'un échantillon des fichiers WSDL que nous avons explorés, c'est que malgré leur appartenance au même domaine (voyages), il n'y a pas de points communs entre eux, aucun type n'a le même nom qu'un autre type d'un autre fichier. Ce fut le cas pour tous les fichiers qu'on a testés. Même la façon de nommer les types n'est pas la même, certains utilisant les soulignes (*underscores*, ex : *Get_MoneyMattersResult*), d'autres pas. Aucune convention ou norme n'est suivi. C'est ainsi qu'*imaginer* un test a été impossible, car comment passer d'un fichier à l'autre, s'il n'y a rien de commun entre les deux, si ce n'est des types primitifs (*String*, *int*, *boolean*, etc.) ?

Les Figure 6.16, Figure 6.20 et Figure 6.24 sont des extraits des fichiers WSDL qui sont mis en annexes :

Premier exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
 targetNamespace="https://192.168.1.230/"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:tns="https://192.168.1.230/"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
 <wsdl:types>
  <s:schema elementFormDefault="qualified"
   targetNamespace="https://192.168.1.230/">
   <s:element name="GetHotelGallery">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="0" name="HotelName" type="s:string"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:element name="GetHotelGalleryResponse">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="0" name="GetHotelGalleryResult">
```

```
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
 </s:schema>
</wsdl:types>
 <wsdl:message name="GetHotelGallerySoapIn">
  <wsdl:part element="tns:GetHotelGallery" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetHotelGallerySoapOut">
  <wsdl:part element="tns:GetHotelGalleryResponse"
name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetHotelGalleryHttpGetIn">
  <wsdl:part name="HotelName" type="s:string"/>
 </wsdl:message>
 <wsdl:message name="GetHotelGalleryHttpGetOut">
  <wsdl:part name="Body"/>
 </wsdl:message>
 <wsdl:message name="GetHotelGalleryHttpPostIn">
  <wsdl:part name="HotelName" type="s:string"/>
 </wsdl:message>
 <wsdl:message name="GetHotelGalleryHttpPostOut">
  <wsdl:part name="Body"/>
 </wsdl:message>
 <wsdl:portType name="TCServicesSoap">
  <wsdl:operation name="GetHotelGallery">
   <wsdl:input message="tns:GetHotelGallerySoapIn"/>
   <wsdl:output message="tns:GetHotelGallerySoapOut"/>
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:portType name="TCServicesHttpGet">
  <wsdl:operation name="GetHotelGallery">
   <wsdl:input message="tns:GetHotelGalleryHttpGetIn"/>
```

```
    <wsdl:output message="tns:GetHotelGalleryHttpGetOut"/>
   </wsdl:operation>
 </wsdl:portType>
 <wsdl:portType name="TCServicesHttpPost">
   <wsdl:operation name="GetHotelGallery">
    <wsdl:input message="tns:GetHotelGalleryHttpPostIn"/>
    <wsdl:output message="tns:GetHotelGalleryHttpPostOut"/>
   </wsdl:operation>
 </wsdl:portType>
    (...)
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Figure 6.16. Fichier WSDL (TCServices) trouvé sur le site d'Aleph suite à la recherche du mot-clé « travel »

Par la suite, nous décortiquons l'exemple. Nous présentons les opérations présentes dans la Figure 6.16 ainsi que leur message d'entrée et le message de sortie, et le type de leurs *parts*.

Exemple: L'opération *GetHotelGallery* a deux messages: *GetHotelGallerySoapIn* et *GetHotelGallerySoapOut*. *GetHotelGallerySoapIn* a un *part* d'element: *GetHotelGallery* et possède un *part* de nom *parameters*. L'*element GetHotelGallery* est de type String. Le même exercice peut être fait pour chacune des opérations présentes dans les Figure 6.17, Figure 6.21, et Figure 6.25.

Opération	Message d'entrée	Message de sortie
GetHotelGallery	`tns:GetHotelGallerySoapIn`	`tns:GetHotelGallerySoapOut`
GetHotelGallery	`tns:GetHotelGalleryHttpGetIn`	`tns:GetHotelGalleryHttpGetOut`

Figure 6.17. Opérations présentes dans le fichier TCServices

Chaque message possède un *part*, qui a 1) un *element* ou un *type* et 2) un *name*.

Message	Élement ou type du *part*	Nom du *part*
tns:GetHotelGallerySoapIn	tns:GetHotelGallery	parameters
tns:GetHotelGallerySoapOut	tns:GetHotelGalleryRespons e	parameters
tns:GetHotelGalleryHttpGetIn	String	HotelName
tns:GetHotelGalleryHttpGetOut	Non précisé	Body

Figure 6.18. Élements des messages des opérations du fichier TCServices

Voici les types et noms des éléments qui n'ont pas un type primitif (référent à un type détaillé dans le fichier WSDL).

Element	Type de l'élément	Nom de l'élément
tns:GetHotelGallery	String	HotelName
tns:GetHotelGalleryResponse	GetHotelGalleryResult	Non précisé

Figure 6.19. Types et noms des elements dont le type n'est pas primitif

Deuxième exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
 targetNamespace="http://tempuri.org/"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:tns="http://tempuri.org/"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

```
(...)

<wsdl:message name="HelloWorldSoapIn">
  <wsdl:part element="tns:HelloWorld" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="HelloWorldSoapOut">
  <wsdl:part element="tns:HelloWorldResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetHotelListingSoapIn">
  <wsdl:part element="tns:GetHotelListing" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetHotelListingSoapOut">
  <wsdl:part element="tns:GetHotelListingResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="MakeReservationSoapIn">
  <wsdl:part element="tns:MakeReservation" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="MakeReservationSoapOut">
  <wsdl:part element="tns:MakeReservationResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="CancelReservationSoapIn">
  <wsdl:part element="tns:CancelReservation" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="CancelReservationSoapOut">
  <wsdl:part element="tns:CancelReservationResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="ViewReservationSoapIn">
  <wsdl:part element="tns:ViewReservation" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="ViewReservationSoapOut">
  <wsdl:part element="tns:ViewReservationResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:portType name="HotelReservationSoap">
  <wsdl:operation name="HelloWorld">
    <wsdl:input message="tns:HelloWorldSoapIn"/>
    <wsdl:output message="tns:HelloWorldSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="GetHotelListing">
    <wsdl:input message="tns:GetHotelListingSoapIn"/>
    <wsdl:output message="tns:GetHotelListingSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="MakeReservation">
```

```
   <wsdl:input message="tns:MakeReservationSoapIn"/>
    <wsdl:output message="tns:MakeReservationSoapOut"/>
   </wsdl:operation>
   <wsdl:operation name="CancelReservation">
    <wsdl:input message="tns:CancelReservationSoapIn"/>
    <wsdl:output message="tns:CancelReservationSoapOut"/>
   </wsdl:operation>
   <wsdl:operation name="ViewReservation">
    <wsdl:input message="tns:ViewReservationSoapIn"/>
    <wsdl:output message="tns:ViewReservationSoapOut"/>
   </wsdl:operation>
  </wsdl:portType>
   <wsdl:port binding="tns:HotelReservationSoap" name="HotelReservationSoap">
    <soap:address
location="http://dotnet.borland.com/ws/Travel/HotelReservation.asmx"/>
   </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 6.20. Fichier WSDL (HotelReservation) trouvé sur le site d'Aleph suite à la recherche du mot-clé « travel »

Nous présentons les opérations présentes dans la Figure 6.20 ainsi que leur message d'entrée et de sortie.

Opération	Message d'entrée	Message de sortie
HelloWorld	`tns:HelloWorldSoapIn`	`tns:HelloWorldSoapOut`
GetHotelListing	`tns:GetHotelListingSoapIn`	`tns:GetHotelListingSoapOut`
MakeReservation	`tns:MakeReservationSoapIn`	`tns:MakeReservationSoapOut`
CancelReservation	`tns:CancelReservationSoapIn`	`tns:CancelReservationSoapOut`
ViewReservation	`tns:ViewReservationSoapIn`	`tns:ViewReservationSoapOut`

Figure 6.21. Opérations présentes dans le fichier HotelReservation

Chaque message possède un part, qui a un *element* ou un *type* et un *name*.

Message	Élement ou type du *part*	Nom du *part*
HelloWorldSoapIn	`tns: HelloWorld`	`parameters`
HelloWorldSoapOut	`tns: HelloWorldResponse`	`parameters`
GetHotelListingSoapIn	`tns: GetHotelListing`	`parameters`
GetHotelListingSoapOut	`tns: GetHotelListingResponse`	`parameters`
MakeReservationSoapIn	`tns: MakeReservation`	`parameters`
MakeReservationSoapOut	`tns: MakeReservationResponse`	`parameters`
CancelReservationSoapIn	`tns: CancelReservation`	`parameters`
tns:CancelReservationSoapOut	`tns: CancelReservationRespons e`	`parameters`
tns:ViewReservationSoapIn	`tns: ViewReservation`	`parameters`
tns:ViewReservationSoapOut	`tns: ViewReservationResponse`	`parameters`

Figure 6.22. Élements des messages des opérations du fichier HoteReservation

Si l'élément est de type complexe, il peut contenir un élément qui possède un nom et un type.

Élement ou type	Nom	Type

HelloWorld	Non précisé	Non precisé
HelloWorldResponse	HelloWorldResult	string
GetHotelListing	HotelId	int
GetHotelListingResponse	GetHotelListingResult	tns: Hotel (*)
MakeReservation	Resv	tns: Reservation (*)
MakeReservationResponse	MakeReservationResult	int
CancelReservation	HotelReservationId	int
CancelReservationResponse	CancelReservationResult	boolean
ViewReservation	CustomerId	int
ViewReservationResponse	ViewReservationResult	tns: Reservation (*)

Figure 6.23. Description des élements contenus dans les messages des opérations du fichier HotelReservation

(*) : Ces types sont eux-mêmes complexes. Leur *arborescence* est détaillée dans le WSDL lui-même.

Troisième exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
 targetNamespace="http://www.anite.com/AniteTravelWS"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:tns="http://www.anite.com/AniteTravelWS"
```

```
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
(...)
 <wsdl:message name="GetFaresSoapIn">
  <wsdl:part element="tns:GetFares" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetFaresSoapOut">
  <wsdl:part element="tns:GetFaresResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetFaresUserAuthentication">
  <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
 </wsdl:message>
 <wsdl:message name="GetFaresWithAvailSoapIn">
  <wsdl:part element="tns:GetFaresWithAvail" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetFaresWithAvailSoapOut">
  <wsdl:part element="tns:GetFaresWithAvailResponse"
name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetFaresWithAvailUserAuthentication">
  <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
 </wsdl:message>
 <wsdl:message name="GetFareAvailSoapIn">
  <wsdl:part element="tns:GetFareAvail" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetFareAvailSoapOut">
  <wsdl:part element="tns:GetFareAvailResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetFareAvailUserAuthentication">
  <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
 </wsdl:message>
 <wsdl:message name="BookFareSoapIn">
  <wsdl:part element="tns:BookFare" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="BookFareSoapOut">
  <wsdl:part element="tns:BookFareResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="BookFareUserAuthentication">
  <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
```

```
</wsdl:message>
 <wsdl:message name="GetAccomAvailSoapIn">
  <wsdl:part element="tns:GetAccomAvail" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetAccomAvailSoapOut">
  <wsdl:part element="tns:GetAccomAvailResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetAccomAvailUserAuthentication">
  <wsdl:part element="tns:UserAuthentication"
name="UserAuthentication"/>
 </wsdl:message>
 <wsdl:message name="GetPriceSummarySoapIn">
  <wsdl:part element="tns:GetPriceSummary" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetPriceSummarySoapOut">
  <wsdl:part element="tns:GetPriceSummaryResponse"
name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetPriceSummaryUserAuthentication">
  <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
 </wsdl:message>
 <wsdl:message name="MakeReservationSoapIn">
  <wsdl:part element="tns:MakeReservation" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="MakeReservationSoapOut">
  <wsdl:part element="tns:MakeReservationResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="MakeReservationUserAuthentication">
  <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
 </wsdl:message>
 <wsdl:portType name="AniteTravelWSSoap">
  <wsdl:operation name="GetFares">
   <documentation
    xmlns="http://schemas.xmlsoap.org/wsdl/">
       Returns all available fares (without travel/journey
availability)
   </documentation>
   <wsdl:input message="tns:GetFaresSoapIn"/>
   <wsdl:output message="tns:GetFaresSoapOut"/>
```

```
</wsdl:operation>
<wsdl:operation name="GetFaresWithAvail">
 <documentation
  xmlns="http://schemas.xmlsoap.org/wsdl/">
      Returns all available fares with travel/journey availability
 </documentation>
 <wsdl:input message="tns:GetFaresWithAvailSoapIn"/>
 <wsdl:output message="tns:GetFaresWithAvailSoapOut"/>
</wsdl:operation>
<wsdl:operation name="GetFareAvail">
 <documentation
  xmlns="http://schemas.xmlsoap.org/wsdl/">
      Returns travel/journey availability for a selected fare
 </documentation>
 <wsdl:input message="tns:GetFareAvailSoapIn"/>
 <wsdl:output message="tns:GetFareAvailSoapOut"/>
</wsdl:operation>
<wsdl:operation name="BookFare">
 <documentation
  xmlns="http://schemas.xmlsoap.org/wsdl/">
      Book/Sell selected fare
 </documentation>
 <wsdl:input message="tns:BookFareSoapIn"/>
 <wsdl:output message="tns:BookFareSoapOut"/>
</wsdl:operation>
<wsdl:operation name="GetAccomAvail">
 <documentation
  xmlns="http://schemas.xmlsoap.org/wsdl/">
      Get accommodation availability
 </documentation>
 <wsdl:input message="tns:GetAccomAvailSoapIn"/>
 <wsdl:output message="tns:GetAccomAvailSoapOut"/>
</wsdl:operation>
<wsdl:operation name="GetPriceSummary">
 <documentation
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
        Returns a summary of prices for all the travel components
added to the reservation so far
    </documentation>
    <wsdl:input message="tns:GetPriceSummarySoapIn"/>
    <wsdl:output message="tns:GetPriceSummarySoapOut"/>
   </wsdl:operation>
   <wsdl:operation name="MakeReservation">
    <documentation
     xmlns="http://schemas.xmlsoap.org/wsdl/">
         Makes (Completes) a reservation and returns a locator
    </documentation>
    <wsdl:input message="tns:MakeReservationSoapIn"/>
    <wsdl:output message="tns:MakeReservationSoapOut"/>
   </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="AniteTravelWSSoap" type="tns:AniteTravelWSSoap">
  <wsdl:service name="AniteTravelWS">
  <wsdl:port binding="tns:AniteTravelWSSoap" name="AniteTravelWSSoap">
   <soap:address location="http://194.203.47.11/AniteTravelWS/AniteTravelWS.asmx"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Figure 6.24. Fichier WSDL (AniteTravelWS) trouvé sur le site d'Aleph suite à la recherche du mot-clé « travel »

On décortique le fichier *AniteTravelWS* de la même manière que pour le fichier précédent.

Opération	Message d'entrée	Message de sortie
GetFares	tns: GetFaresSoapIn	tns: GetFaresSoapOut
GetFaresWithAvail	tns: GetFaresWithAvailSoapIn	tns: GetFaresWithAvailSoapOut
GetFareAvail	tns: GetFareAvailSoapIn	tns: GetFareAvailSoapOut

BookFare	tns: BookFareSoapIn	tns: BookFareSoapOut
GetAccomAvail	tns: GetAccomAvailSoapIn	tns: GetAccomAvailSoapOut
GetPriceSummary	tns: GetPriceSummarySoapIn	tns: GetPriceSummarySoapOut
MakeReservation	tns: GetMakeReservationSoapIn	tns: GetMakeReservationSoapOut

Figure 6.25. Opérations présentes dans le fichier AniteTravelWS

Les messages sont composés des éléments suivants.

Message	Élement ou type du *part*	Nom du *part*
GetFaresSoapIn	tns: GetFares	parameters
GetFaresSoapOut	tns: GetFaresResponse	parameters
GetFaresWithAvailSoapIn	tns: GetFaresWithAvail	parameters
GetFaresWithAvailSoapOut	tns: GetFaresWithAvailResponse	parameters
GetFarAvailSoapIn	tns: GetFareAvail	parameters
GetFarAvailSoapOut	tns: GetFareAvailResponse	parameters
BookFareSoapIn	tns: BookFare	Parameters
BookFareSoapOut	tns: BookFareResponse	Parameters

GetAccomAvailSoapIn	tns: GetAccomAvail	parameters
GetAccomAvailSoapOut	tns: GetAccomAvailResponse	parameters
GetPriceSummarySoapIn	tns: GetPriceSummary	parameters
GetPriceSummarySoapOut	tns: GetPriceSummaryResponse	parameters
MakeReservationSoapIn	tns: MakeReservation	parameters
MakeReservationSoapOut	tns: MakeReservationResponse	parameters

Figure 6.26. Élements des messages des opérations du fichier HoteReservation

Les éléments complexes se décomposent comme suit.

Élement ou type	Nom	Type
GetFares	XMLRequestDoc	Non precisé
GetFaresResponse	GetFaresResponse	Non precisé
GetFaresWithAvail	XMLRequestDoc	Non précisé
GetFaresWithAvailResponse	GetFaresWithAvailRespons e	Non précisé
GetFareAvail	XMLRequestDoc	Non précisé
GetFareAvailResponse	GetFareAvailResult	Non précisé
BookFare	XMLRequestDoc	Non précisé
BookFareResponse	BookFareResult	Non precise

GetAccomAvail	XMLRequestDoc	Non précisé
GetAccomAvailResponse	GetAccomAvailResult	Non précisé
GetPriceSummary	XMLRequestDoc	Non précisé
GetPriceSummaryResponse	GetPriceSummaryResult	Non précisé
MakeReservation	XMLRequestDoc	Non précisé
MakeReservationResponse	MakeReservationResult	Non précisé

Figure 6.27. Description des élements contenus dans les messages des opérations du fichier HotelReservation

Nous présentons dans la section suivante nos remarques après l'analyse de ces fichiers.

6.3.2.2 Analyse

Les trois fichiers *TCServices, HotelReservation*, *AniteTravelWS* sont des fichiers WSDL trouvés grâce à une recherche par mot-clé. Par conséquent, ils sont du même « domaine ». L'analyse faite sur les attributs de ces fichiers amène aux remarques suivantes.

- On ne précise pas toujours le type de l'attribut *element* : manque de constance, ou manque de normes pour la description des fichiers WSDL.
- Beaucoup des *parts* contenus dans les messages ont le même nom : *parameters*.

La remarque la plus importante à notre sens est cependant la suivante.

L'attribut *element* des messages d'entrée et de sorties des opérations, dans cet échantillon de trois fichiers qu'on a choisi, n'a jamais le même nom de fichier en fichier. Nous nous basions essentiellement sur un appariement de ces noms de types car ce sont des documents d'affaire. Si le nom du type n'est jamais le même, il est impossible de faire de

l'appariement de type basé sur les noms. Il reste à explorer l'appariement de signatures basé sur la structure (*cf* le chapitre suivant).

Ainsi, les noms des entrées et sorties des opérations n'étant pas standards, même si des opérations satisfont le même besoin (offrent le même service), leurs entrées et sorties sont nommés différemment. Il est par conséquent impossible de faire de l'appariement de signature, si la méthode d'appariement se base sur le nom des entrées et sorties.

6.3.2.3 Conclusion

Si les noms (ou la façon de nommer) des éléments des fichiers WSDL ne sont pas normalisés, il n'y a aucun moyen de faire de l'appariement des signatures basé sur les noms des parts (ou types). Les résultats dans ce cas sont totalement insatisfaisants.

6.3.3. Troisième évaluation

Suite à notre expérience avec le site Aleph, nous avons décidé de chercher d'autres fichiers WSDL pour faire nos tests. C'est ainsi que nous avons décidé d'utiliser ceux du *Web Service Challenge*. Sur ce site, plusieurs fichiers sont offerts. Les solutions à trouver sont présentées dans un fichier (pdf ou texte) (Figure 6.28).

```
1.      purchaseALT.wsdl->reserveRental.wsdl-> reserveRoom.wsdl-
>createItinerary.wsdl
2.      findMostRelevantStocks->getStockPriceMany-
>performStockResearch->purchaseOptimalStock
3.      findMostRelevantStock->getStockPrice->purchaseStock
4.      getStockPrice->purchaseStock
5.      purchaseFlowers
6.      findCloseAttorney->scheduleAttorney
7.      findCloseLibrary->findISBN->checkOutLibraryBook
8.      findCloseVideoStore->rentVideo
9.      findThemePark->purchaseThemeParkTicket
10.     getForecast
11.     findCloseFlorist->purchaseFlowers
12.     findDate->scheduleDate->findDateRestaurant-
>reserveDateRestaurant
```

```
13.     findCloseMountain->purchaseLiftTicket
14.     findCloseVeterinarian->scheduleVeterinarian
15.     scheduleShipment
```

Figure 6.28. Résultats de composition escomptés dans le Web Service Challenge 05 (WSC)

Dans un autre fichier, on nous fournit les données disponibles (*provided*) et les sorties escomptées (*Resultant*):

```
<CompositionRoutine>
                <Provided> partname1, partname2, partname3
</Provided>
                <Resultant> partname1, partname2, partname3
</Resultant>
</CompositionRoutine>
```

Notre défi était simple : tester notre programme sur ces fichiers.

6.3.3.1 Résultats des tests de la troisième évaluation

Ces derniers tests nous ont permis de corriger quelques imperfections de notre programme : ils couvrent beaucoup de cas plausibles de composition de services web. Par contre, il est important de souligner que quelques tests n'étaient pas possibles pour une raison simple : des données manquaient dans les données d'entrées. Les concepteurs de ces tests auraient omis de les fournir.

Exemple de tests réussis :

Test1 :

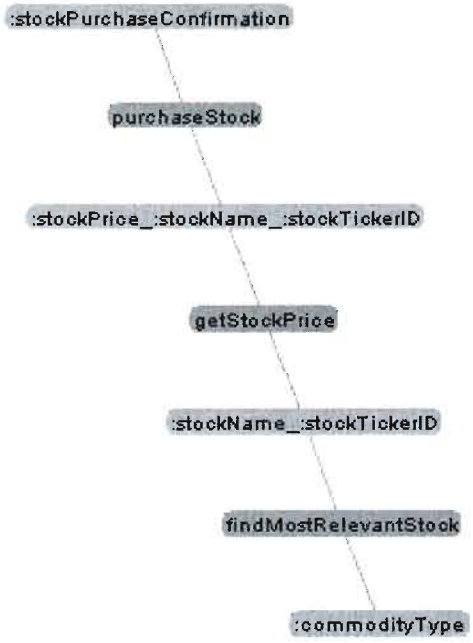
Dans cet exemple, nous avions les données suivantes :

```
<CompositionRoutine>
                <Provided>commodityType</Provided>
                <Resultant> stockPurchaseConfirmationOptimal
</Resultant>
</CompositionRoutine>
```

Le résultat escompté était le suivant :

```
. findMostRelevantStocks->getStockPriceMany->performStockResearch-
>purchaseOptimalStock
```

Voici le résultat trouvé par notre programme :



Le test est réussi puisqu'on a pu avoir en sortie stockPurchaseConfirmationOptimal et ce, en passant dans l'ordre par les opérations *findMostRelevantStocks*, *getStockPriceMany*, *performStockResearch* et *purchaseOptimalStock.*

Test 2 :

Entrées :

```
<CompositionRoutine>
                <Provided>
pickupLocationName, pickupLocationID, firstName, lastName,
middleInitial, custStreetAddress, custCityAddress, custStateAddress,
custZipAddress
                </Provided>
                <Resultant> shipmentTrackingNumber, shipmentCost
</Resultant>
</CompositionRoutine>
```

Résultats escomptés :

```
scheduleShipment
```

Résultat trouvé par notre programme :

Dans ce cas précis, notre programme a trouvé la bonne opération. Cette opération retourne un élément de plus que ceux escomptés : *arrivalDateTime*. Ceci n'enlève rien au succès du test puisque nous avons trouvé ce que cherchions : *shipmentCost* et *shipmentTrackingNumber*, en utilisant l'opération *scheduleShipment*.

Certains tests n'ont pas marché car il manquait des données en entrée. C'est le cas pour le test 3 et le test 4 qui sont exposés dans ce qui suit.

Test 3 :

Entrées :

```
<CompositionRoutine>
        <Provided>
                custStreetAddress,custCityAddress,
                custStateAddress,custZipAddress,dealerType
        </Provided>
        <Resultant> attorneyConfirmation  </Resultant>
</CompositionRoutine>
```

Résultat escompté :

```
findCloseAttorney->scheduleAttorney
```

Les fichiers WSDL *findCloseAttorney. Wsdl* et *scheduleAttorney.wsdl* offerts contiennent deux opérations :

- L'opération *findCloseAttorney* : prend en entrée un message dont les *parts* sont *custStreetAddress, custCityAddress, custStateAddress* et *custZipAddress* et retourne en sortie un message dont les parts sont *attorneyName*, *attorneyID*, *attorneyStreetAddress, attorneyCityAddress, attorneyStateAddres, attorneyZipAddres, attorneyPhone.*

- L'opération *scheduleAttorney* : prend en entrée un message dont les *parts* sont *custFirstName*, *custLastName*, *custMiddleInitial*, *attorneyName*, *attorneyID*, *attorneyStreetAddress*, *attorneyCityAddress*, *attorneyStateAddress*, *attorneyZipAddress*, et *arrivalDateTime* et retourne en sortie un message dont le *part* est *attorneyConfirmation.*

Explications :

Les *parts* que nous avons en entrée nous permettent d'invoquer l'opération *findCloseAttorney*, puisqu'ils contiennent *custStreetAddress, custCityAddress, custStateAddress* et *custZipAddress.* Cependant, il est impossible d'invoquer l'opération *scheduleAttorney* puisque les *parts custFirstName*, *custLastName*, *custMiddleInitial*, et *arrivalDateTime* ne sont pas disponibles et ce, même dans les *parts* retournés en sortie par *findCloseAttorney.*

Test 4 :

Entrées :

```
<CompositionRoutine>
        <Provided>

custStreetAddress,custCityAddress,custStateAddress,custZipAddress
        </Provided>
        <Resultant> libraryName, checkOutLibraryBookConfirmation
</Resultant>
</CompositionRoutine>
```

Résultat escompté :

```
findCloseLibrary->findISBN->checkOutLibraryBook
```

Les fichiers WSDL *findCloseLibrary.wsdl, findISBN.wsdl et checkOutLibraryBook.wsdl* offerts contiennent trois opérations :

- L'opération *findCloseLibrary* : prend en entrée un message dont les *parts* sont *custStreetAddress, custCityAddress, custStateAddress* et *custZipAddress* et retourne en sortie un message dont les parts sont *LibraryName, LibraryStreetAddress, LibraryCitytAddress, LibraryStateAddress et LibraryZipAddres.*

- L'opération *findISBN* : prend en entrée un message dont le *part* est *bookTitle* et retourne en sortie un message dont les *parts* sont *bookISBN et bookTitle.*

- L'opération *checkOutLibraryBook* : prend en entrée un message dont les *parts* sont *bookISBN et libraryName* et retourne en sortie un message dont les *parts* sont *libraryName et checkOutLibraryBookConfirmation.*

Explications :

Les *parts* que nous avons en entrée nous permettent d'invoquer l'opération *findCloseLibrary*, puisqu'ils contiennent *custStreetAddress, custCityAddress, custStateAddress* et *custZipAddress*. Cependant, il est impossible d'invoquer l'opération *findISBN* puisque le part *bookTitle* n'est pas disponible, ni dans les entrées, ni dans les sorties de l'opération *findCloseLibrary*. Même chose pour l'opération *checkOutLibraryBook,* puisque le *part* libraryName n'est pas disponible ni dans les entrées ni dans les sorties de l'opération *findCloseLibrary*.

6.3.3.2 Conclusion

Sur les fichiers du *Web Service Challenge* notre approche s'est avérée concluante toutes les fois qu'une solution était possible. Le problème de complexité s'est posé quand en entrée, on nous fournissait beaucoup (plus de dix) données en entrée. (trouver tous les sous-ensembles d'un ensemble de dix éléments peut être laborieux). Sur les quinze cas testés, quatre compositions étaient impossibles à réaliser à cause des données manquantes (même cas que les tests 3 et 4), quatre compositions prenaient beaucoup de temps à trouver la solution. Dans sept cas, nous avons trouvé sans problèmes la solution escomptée.

CHAPITRE 7

ALGORITHMES D'APPARIEMENT DE TYPE

L'approche de l'appariement par équivalence de noms n'a pas été concluante, essentiellement parce que la façon de nommer les types des éléments n'est pas uniforme entre un fichier WSDL et un autre, et ce même s'ils font partie du même domaine. Nous présentons ici une autre approche pour l'appariement de type qu'il serait intéressant d'explorer dans le futur.

7.1. Principes

Les langages de programmation s'attaquent au problème d'appariement de type de deux manières différentes. Les langages fortement typés, comme Ada ou Java, utilisent la stratégie nommée *name equivalence*. Tandis que les autres langages, comme le langage C, optent pour la méthode nommée *structural equivalence*. La méthode *name equivalence* est assez faible, pour la simple raison que différents fournisseurs de services pourraient utiliser différentes appellations pour le même service ou encore la même appellation pour des services différents (c'est ce que nous avons vu dans le chapitre précédent). Ce problème est assez répandu dans le domaine des technologies distribuées et il a été traité des deux façons suivantes (Mili et al., 2004):

- L'approche dite *verticale* : Dans cette approche, dans chaque secteur d'activités, les concepts communs sont standardisés. Plusieurs groupes dans

l'industrie dont *Open Travel Alliance* (*http://opentravel.org/*) ont adopté cette approche. En effet, *OTA* a standardisé les messages en relation avec l'industrie du tourisme, et a, par cet effet, assuré l'interopérabilité entre les différents membres de cette industrie (qui adhèrent à leur standard). Il y a aussi *RosettaNet*, ou encore *ebXML*, qui ont standardisé les processus inter-organisationnels, ainsi que les documents échangés par ces mêmes processus.

- L'approche dite *ontologique* : Cette approche définit des langages pour décrire des concepts, au lieu de proposer une liste restreinte de concepts dont la description et le déchiffrement sont seulement possibles par les humains. Des initiatives comme OWL et DAML+OIL sont de bons exemples de l'application de cette approche. Les langages ontologiques représentent les concepts en utilisant une liste prédéfinie de constructeurs, ainsi qu'un vocabulaire partagé, bien qu'il soit de bas niveau. Ils permettent aux outils automatisés de faire un lien (exemple : l'équivalence) entre les concepts.

Nous voyons la différence entre les deux approches comme une différence de niveaux d'abstraction. Les langages ontologicaux peuvent établir des relations entre des concepts de haut niveau, en autant que ces concepts soient exprimés en utilisant un vocabulaire commun. L'approche verticale, elle, considère le vocabulaire commun lui-même comme le concept de haut niveau.

7.2. Mise en oeuvre

Il est possible de mettre en oeuvre des algorithmes fondés sur les deux stratégies (*name equivalence*, *structural equivalence*). La recherche basée sur la composition de services web commencerait en utilisant la recherche basique qui comparerait deux noms de services (*name equivalence*). Si la recherche ne donne aucun résultat satisfaisant, on ferait appel aux algorithmes basés sur l'approche structurale.

Nous avons présenté l'algorithme fondé sur l'équivalence des noms dans le chapitre 6. Nous présentons dans ce qui suit une autre approche d'équivalence des types.

7.2.1. Une autre approche possible

Dans cette approche, la structure de l'arbre n'est pas ce qui nous intéresse le plus, ce sont plutôt les feuilles de l'arbre qui suscitent notre intérêt. Dans la Figure 7.1, on présente un exemple où une telle équivalence serait souhaitable :

```
<complexType name="Address">
        <sequence>
                <element
                name="streetNumber"
                        type="xs:string"/>
                <element name="streetName"
                        type="xs:string"/>
                <element name="city"
                        type="xs:string"/>
                <element name="zipCode"
                        type="xs:string"/>
        </sequence>
</complexType>
<complexType name="Order">
        <sequence>
                <element
                name="billingAddress"
                        type="ns:Address"/>
                <element
                name="shippingAdress"
                        type="ns:Address">

        </sequence>
</complexType>
```

```
<complexType name="Order">
        <sequence>
                <element
                name="shippingStreetNumber"
                        type="xs:string"/>
                <element
                name="shippingStreetName"
                        type="xs:string"/>
                <element name="shippingCity"
                        type="xs:string"/>
                <element
                name="shippingZipCode"
                        type="xs:string"/>
                <element
                name="billingStreetNumber"
                        type="xs:string"/>
                <element
                name="billingStreetName"
                        type="xs:string"/>
                <element name="billingCity"
                        type="xs:string"/>
                <element name="billingZipCode"
                        type="xs:string"/>
        </sequence>
```

Figure 7.1. Deux schémas avec contenus de données équivalents et des structures différentes (Mili et al.)

7.2.1.1 Explications

Dans le premier tableau (à gauche), l'arbre a deux niveaux de profondeur : deux types complexes sont représentés (*billingAddress* et *shippingAddress*), qui sont eux-mêmes de types *Address*. Ce dernier type est défini par quatre attributs : *streetNumber*, *streetName*, *city* et *zipCode* qui sont tous de types string. Nous avons, par conséquent, deux types complexes de quatre attributs (de type string). Dans le deuxième tableau (à droite), l'arbre est d'un seul niveau de profondeur : on définit un type complexe avec huit attributs (de type string aussi).

Les deux tableaux définissent des types équivalents sémantiquement, mais de profondeur différente. Il serait souhaitable, grâce à un algorithme d'équivalence fondé sur les feuilles des deux arbres, d'apparier ces deux types qui sont, en pratique, équivalents.

CHAPITRE 8

CONCLUSION

Aujourd'hui, bon nombre d'entreprises sont représentées sur le net. Souvent, elles y basent même l'essentiel de leurs activités économiques. Grâce à Internet, ces entreprises sont capables d'échanger des données à distance et de les invoquer. En particulier, les services web permettent une communication facile et fluide entre les entreprises et leurs partenaires et clients. UDDI permet l'enregistrement d'un service web et son invocation par un utilisateur (*consumer*). Mais un seul service web peut ne pas suffire à tous nos besoins. Pour combler certains besoins du client, il est nécessaire de composer des services web. La combinaison des services qu'ils offrent permettra d'en faire un flux de services web comblant le besoin du client. La composition de services web est un sujet qui a connu beaucoup d'intérêt dans les dernières années. L'approche que les chercheurs ont prise pour composer des services web varie. L'une des approches est l'approche statique où *les composants sont choisis avant d'être compilés et déployés* (Dustdar et Schreiner, 2005). Néanmoins, l'intervention de l'être humain, parfois laborieuse, est essentielle dans ce cas. L'autre approche est l'approche dynamique où *les services web à composer sont choisis au moment de l'exécution, durant laquelle une recherche des services est effectuée dans les registres* (Channa et al., 2005). Cette approche n'a pas besoin de l'intervention humaine mais nécessite une manipulation des fichiers WSDL ou le rajout d'une couche sémantique à UDDI, comme pour StarWSCOP, (Dustdar et Schneider, 2005).

Dans ce travail, nous avons proposé une approche distincte de composition de services web. Notre composition de services web se fait à partir d'une recherche basée sur

l'appariement des types. Notre approche a l'avantage de n'exiger aucune modification des standards (UDDI, SOAP, etc.) et aucun rajout d'attributs aux fichiers WSDL. Ainsi, il n'y a ni effort de mise à jour ou de migration de données ni souci de compatibilité de notre programme avec un quelconque autre outil.

Nous avons testé notre algorithme d'appariement des types basé sur les noms sur des exemples créés localement, puis sur des fichiers WSDL importés du site *Aleph*. Ensuite, nous avons proposé une autre manière de faire de l'appariement de types dans le chapitre 7.

Notre approche n'a pas fonctionné car la façon de nommer les types des éléments des fichiers WSDL n'est pas uniforme, les noms des entrées et des sorties des opérations n'est généralement pas normalisé, et ce, même si on utilise les références indexées dans notre recherche, i.e. même si on filtre les fichiers WSDL par industrie ou domaine, avant d'appliquer notre programme de composition de services web. Par conséquent, si aucune normalisation des noms n'est faite dans le commerce électronique, notre approche ne pourra donner de résultats satisfaisants.

Nous proposons comme travaux futurs d'explorer la méthode d'appariement vue dans le chapitre 7. Le défi serait d'atteindre des résultats avec le moins de « *false positives* » (Cf. 2.4.1) possibles.

Annexes

Annexe A

Code source de la méthode EnumerateRealizations

```
public static Realization enumerateRealizations(int stageNumber,
                                                 Collection previousF,
                                                 Collection previousT,
                                                 Collection availableOperations,
                                                 Collection partParents){

        // in the algorithm outPutParts is Fk-1, setOfTypes is Tk-2
        Collection currentT            = null;
        Collection firstUnion          = null;
        Collection secondUnion         = null; // a temporary collection built on the
                                       union of two collections
        Collection newSubsets  = null;         // the collection of the new subsets
        Collection firstCommonElements = null;// a collection built on the common
                                       elements between two collections
        Collection secondCommonElements = null;// a collection built on the common
                                                 elements between two collections
        Collection restOfElements       = null;// a collection built by removing
                                                 certain elements from a collection
        Collection nextOperations       = null;// a collection of operations
        Collection newOperations        = null; // the new reachable operations
        Map outPutPreviousF ;          // the outputs of the collection previousF
        Map nextInputs;                          // the inputs of the next subset
        Map nextOutputs;                         // the outputs of the next subset
        Collection subsetsNewOperations = null;// the subsets of the new reached
                                                          operations
        Collection nodes                = null;
        Collection nextSubsetNewOperations     = null;
        Collection tempOutputs                 = null;
        Part tempPart;
        Collection operationParents            = new ArrayList();
        Collection operationChildren           = new ArrayList();
        Collection partChildren                = new ArrayList();
        outPutPreviousF                        = getOutPuts(previousF);

        //Tk-1 <- Tk-2 U Output (Fk-1);
```

```
// currentT is the union of both collections = Tk-1 in the algorithm
currentT = constructUnion(previousT, outPutPreviousF.values());
Collection temp = getTheParts(currentT);
RealizationNodePart currentPartNode = null;
if(!belongsNode(temp, nodeParts)){
        currentPartNode = new RealizationNodePart(partialRealization,
                        outPutPreviousF.values(), partParents);
        nodes = partialRealization.getRealizationNodes();
        nodes.add(currentPartNode);
        partialRealization.setRealizationNodes(nodes);
        nodeParts.add(currentPartNode);
}
else{
        currentPartNode =
                (RealizationNodePart)nodeParts.get(nodeParts.size()-1);
}
operationParents.add(currentPartNode);

//Ck-1 <- {f | Input (f) is a subset of Tk-1 but not of Tk-2};

//the new subsets are the subset that belong to currentT but not to
previousT
newSubsets = notCommonSubsets(currentT, previousT);

// the new operations are the operations that we can reach with the
        newSubsets
newOperations = getNewOperations(availableOperations,newSubsets );
// the operations that we just reached are this part's children

// when the second argument is 1, we're trying to get subsets for
//operations
// subsetsNewOperations are the subsets of the new operations that we can
//reach
subsetsNewOperations = Subsets.getSubSets(newOperations,1,subsetsLimit);

// to iterate on the subsets of the new operations
Iterator iterSubsets = subsetsNewOperations.iterator();

// for every subset H of Ck-1 do
while(iterSubsets.hasNext()){
        Collection newPartParents = new ArrayList();
        operationChildren = new ArrayList();

        // get next collection of operations in the subsets
        nextSubsetNewOperations = (Collection)iterSubsets.next();
```

```
// get the input parts of the collection nextSubset
nextInputs = getInputs(nextSubsetNewOperations);

// get the output parts of the collection nextSubset
nextOutputs = getOutPuts(nextSubsetNewOperations);

// firstCommonElements = outPut(g) INTER outPut(Fk-1)
firstCommonElements = getCommonElements(desiredOutPuts,
        outPutPreviousF.values());

// firstUnion = inPut (H) U outPut(g) INTER outPut(Fk-1)
firstUnion =
constructUnion(nextInputs.values(),firstCommonElements);

//restOfElements = inPut (H) U outPut(g) INTER outPut(Fk-1) - Tk-2
restOfElements = removeElements(firstUnion, previousT);

// if outPutPreviousF is a minimal cover of restOfElements
if (isMinimalCover(outPutPreviousF.values(), restOfElements)){

        nextOperations =
                getTheOperations(nextSubsetNewOperations);
        // if a node was not already created with the collection
        //nodeOperations
        if(!belongsOperation (nextOperations, nodeOperations)){
                nodeOperations.addAll(nextOperations);
                // create a new realization node, and add it to
                //the partial realization
                RealizationNodeOperation nodeOperation =
                partialRealization.buildNodeOperation
                        (partialRealization,
                        nextSubsetNewOperations,
                        stageNumber, operationParents);
                partChildren.add(nodeOperation);
                newPartParents.add(nodeOperation);
                currentPartNode.setPartChildren(partChildren);

                //nodeOperation.setOperationParents(currentT);
                Collection tempParents = new ArrayList();
                RealizationNodePart nextPartNode;
                tempOutputs =
                getOutPuts(nextOperations).values();
                if(!belongsNode(tempOutputs, nodeParts)){
                        nextPartNode =
                        new RealizationNodePart
                                (partialRealization,
```

```
                                        tempOutputs,
                                        newPartParents);
                                nodes = partialRealization.
                                        getRealizationNodes();
                                nodes.add(nextPartNode);
                                partialRealization.
                                        setRealizationNodes(nodes);
                                nodeParts.add(nextPartNode);
                        }
                        else{
                                nextPartNode = findNode(tempOutputs,
                                                nodeParts);
                        }
                        operationChildren.add(nextPartNode);
                        nodeOperation.setOperationChildren
                                        (operationChildren);
                        tempParents = nextPartNode.getPartParents();
                        Collection operations
                                = nodeOperation.getOperations();
                        if(!belongsOperations(operations,
                                                tempParents)){
                                Collection partNodeParents =
                                nextPartNode.getPartParents();
                                partNodeParents.add(nodeOperation);
                                nextPartNode.setPartParents
                                                (partNodeParents);
                        }
                }
                // secondUnion = Tk-1 U Output (H)
                secondUnion = constructCollection (currentT,
                                        nextOutputs.values());
                // if Tk-1 U Output (H) contains Output (g) then
                if(contains(secondUnion,desiredOutPuts)){
                        // secondCommonElements  = Output (g) INTER
                        //Output (H)
                        secondCommonElements  =
                                getCommonElements(desiredOutPuts,
                                        nextOutputs.values());
                        // secondCommonElements  = Output (g) INTER
                        //Output (H) - Tk-1
                        secondCommonElements =
                        removeElements(secondCommonElements, currentT);
                        nextOperations = new ArrayList();

                        // if H is a minimal cover of Output (g) INTER
                        //Output (H) - Tk-1 then
```

```
                        if(isMinimalCover(nextOutputs.values(),
                                        secondCommonElements )){


                                //display the realization
                                allRealizations.add
                                        (partialRealization);
                                displayElements(partialRealization);
                                displayNodeOperationsByStage
                                        (partialRealization);
                                // add this realization to the other
                                //realizations
                                allRealizations.add(previousF);
                        }
                }
                //else if output(H) is not contained in Tk-1
                //then make a recursive call
                else if (!contains(currentT, nextOutputs.values())){
                        nextOperations = getTheOperations
                                        (nextSubsetNewOperations);
                        Realization copy = partialRealization.copy();
                        allPartialRealizations.add(copy);
                        enumerateRealizations(stageNumber +
                                1, nextOperations, currentT,
                                desiredOutPuts, partialRealization,
                                availableOperations, partParents);
                }
        }
    }
    return partialRealization;
}
```

Annexe B

Code source des méthodes permettant d'atteindre une réalisation à partir du résultat de la méthode EnumerateRealizations

```
public static HashMap getFunctions(Realization realization, Operation operation){
```

```
Output output;
Collection parts    = new ArrayList();
output              = operation.getOutput();
Part nextPart;
Message message     = output.getMessage();
parts               = message.getParts().values();
HashMap mainMap     = new HashMap();
HashMap secMap      = new HashMap();
mainMap.put(0, secMap);
Collection nodeOperations = new ArrayList();
Iterator tempIter;
Collection operationsAtStage          = new ArrayList();
Collection firstWave                  = new ArrayList();
int longestWave                       = getLongestWave(realization);
String name                           = null;
Iterator iterParts                    = parts.iterator();
firstWave.add(operation);
while(iterParts.hasNext()){
        nextPart = (Part)iterParts.next();
        name = nextPart.getTypeName().getLocalPart().toString();
        secMap.put(name, firstWave);
}
for(int k =1; k <= longestWave; k++){
        //get the operations' node at the stage k-1
        nodeOperations =
        (Collection)realization.getNodeOperationsByStageNumber().get(k-1);
        //iterate on the operations' node
        tempIter = nodeOperations.iterator();
        Iterator iterAtStage;
        RealizationNodeOperation nodeOperation;
        // As long as there is more nodes of operations
        while(tempIter.hasNext()){
                //get the next operation's node
                nodeOperation =
                        (RealizationNodeOperation)tempIter.next();
                EnumerateRealization.displayOperations
                                (nodeOperation.getOperations());
                // get all the operations at that stage
                operationsAtStage = nodeOperation.getOperations();
                iterAtStage = operationsAtStage.iterator();
                Operation nextOperation;
                Part tempPart;
                Collection outputs          = new ArrayList();
                Collection operations       = new ArrayList();
                Operation foundOperation    = null;
                while(iterAtStage.hasNext()){
```

```
                                nextOperation = (Operation)iterAtStage.next();
                                outputs =
                                nextOperation.getOutput().getMessage().
                                        getParts().values();
                                Iterator iterOutputs = outputs.iterator();
                                while(iterOutputs.hasNext()){
                                        tempPart  = (Part)iterOutputs.next();
                                        name      = tempPart.getTypeName().
                                                    getLocalPart().toString();
                                        if(secMap.containsKey(name)){
                                                    foundOperation =
                                                    (Operation)secMap.
                                                            get(tempPart);
                                        }
                                        operations = getUnion(foundOperation,
                                                    nextOperation);
                                        secMap.put(name, operations);
                                }

                        }
                }
        }
        return mainMap;
}
```

```
private static  int getLongestWave(Realization realization){
        int longestWave = 0;
        int temp = 0;
        Collection realizationNodes = realization.getRealizationNodes();
        Iterator iterNodes = realizationNodes.iterator();
        while(iterNodes.hasNext()){
                RealizationNode node = (RealizationNode)iterNodes.next();
                if(node instanceof RealizationNodeOperation){
                        temp =
                        ((RealizationNodeOperation)node).getStageNumber();
                        if(longestWave <= temp){
                                longestWave = temp;
                        }
                }
        }
        return longestWave;
}
```

```
public static ArrayList buildTriplets(Realization realization){
        int longestWave = getLongestWave(realization);
        ArrayList suite = new ArrayList();
        Collection nextWaveOperations = new ArrayList();
        Collection tempOperations = new ArrayList();
        Collection inputs = new ArrayList();
        Collection tempColl = new ArrayList();
        Iterator iterNodeOperations, iterOperations, iterInputs;
        RealizationNodeOperation nextNodeOperation;
        Operation nextOperation;
        String name = null;
        Part nextPart;
        for(int k = 0; k < longestWave; k++){
                nextWaveOperations =
                (Collection)realization.getNodeOperationsByStageNumber().get(k);
                iterNodeOperations = nextWaveOperations.iterator();
                while(iterNodeOperations.hasNext()){
                        nextNodeOperation =
                        (RealizationNodeOperation)iterNodeOperations.next();
                        tempOperations = nextNodeOperation.getOperations();
                        iterOperations = tempOperations.iterator();
                        while(iterOperations.hasNext()){
                                nextOperation =
                                        (Operation)iterOperations.next();
                                inputs =
                                nextOperation.getInput().getMessage().
                                                getParts().values();
                                iterInputs = inputs.iterator();
                                while(iterInputs.hasNext()){
                                        nextPart = (Part)iterInputs.next();
                                        name =
                                        nextPart.getTypeName().getLocalPart()
                                                        .toString();
                                        tempColl =
                                        addElements(k,nextOperation, name);
                                        suite.add(tempColl);
                                }
                        }
                }
        }
        return suite;
}

public static void enumerateImplementations(int position, ArrayList suite,
```

```
                                                HashMap mainMap, Operation[] tab){
        int length          = suite.size();
        HashMap secMap      = (HashMap)mainMap.get(0);
        int k               = 0;
        Operation operation = null;
        String name         = null;
        ArrayList tempColl = new ArrayList();
        Collection operations = new ArrayList();
        Operation tempOperation = null;
        if(position <= length){
                tempColl = (ArrayList)suite.get(position-1);
                k = ((Integer)tempColl.get(0)).intValue();
                name = (String)tempColl.get(2);
                operations = (Collection)secMap.get(name);
                Iterator iterOper = operations.iterator();
                while(iterOper.hasNext()){
                        tempOperation = (Operation)iterOper.next();
                        tab[position-1] = tempOperation;
                        enumerateImplementations(position+1, suite, mainMap,
                                                                        tab);
                }
        }
        else{
                printImplementations(length, suite, tab);
        }
}
```

```
private static void printImplementations(int length, ArrayList suite,
                                                        Operation[] tab){
        ArrayList tempColl = new ArrayList();
        int k =0;
        String name = null;
        Operation operation = null;
        for(int i =0; i<length; i++){
                tempColl = (ArrayList)suite.get(i);
                k = ((Integer)tempColl.get(0)).intValue();
                operation = (Operation)tempColl.get(1);
                name = (String)tempColl.get(2);
                System.out.println("À la phase "+k+ " , le type " +name+"  utilisé
                        par la fonction " +operation.getName()+ "  provient de
                        la fonction : " + tab[i]);
                }
        return;
}
```

```
private static Collection addElements(int k, Operation operation, String name){
        Collection elements = new ArrayList();
        elements.add(new Integer(k));
        elements.add(operation);
        elements.add(name);
        return elements;
}
```

Annexe C

Fichiers WSDL utilisés dans la section 6.3.2.1

Fichier *TCServices*:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
 targetNamespace="https://192.168.1.230/"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
```

```
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:tns="https://192.168.1.230/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<wsdl:types>
 <s:schema elementFormDefault="qualified"
  targetNamespace="https://192.168.1.230/">
  <s:element name="GetHotelGallery">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="HotelName" type="s:string"/>
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="GetHotelGalleryResponse">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="GetHotelGalleryResult">
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
 </s:schema>
</wsdl:types>
<wsdl:message name="GetHotelGallerySoapIn">
 <wsdl:part element="tns:GetHotelGallery" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetHotelGallerySoapOut">
 <wsdl:part element="tns:GetHotelGalleryResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetHotelGalleryHttpGetIn">
 <wsdl:part name="HotelName" type="s:string"/>
</wsdl:message>
<wsdl:message name="GetHotelGalleryHttpGetOut">
 <wsdl:part name="Body"/>
</wsdl:message>
<wsdl:message name="GetHotelGalleryHttpPostIn">
 <wsdl:part name="HotelName" type="s:string"/>
</wsdl:message>
<wsdl:message name="GetHotelGalleryHttpPostOut">
 <wsdl:part name="Body"/>
```

```
 </wsdl:message>
 <wsdl:portType name="TCServicesSoap">
  <wsdl:operation name="GetHotelGallery">
   <wsdl:input message="tns:GetHotelGallerySoapIn"/>
   <wsdl:output message="tns:GetHotelGallerySoapOut"/>
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:portType name="TCServicesHttpGet">
  <wsdl:operation name="GetHotelGallery">
   <wsdl:input message="tns:GetHotelGalleryHttpGetIn"/>
   <wsdl:output message="tns:GetHotelGalleryHttpGetOut"/>
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:portType name="TCServicesHttpPost">
  <wsdl:operation name="GetHotelGallery">
   <wsdl:input message="tns:GetHotelGalleryHttpPostIn"/>
   <wsdl:output message="tns:GetHotelGalleryHttpPostOut"/>
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:service name="TCServices">
  <wsdl:port binding="tns:TCServicesSoap" name="TCServicesSoap">
   <soap:address location="http://www.travel-
center.com/webservices/TCServices.asmx"/>
  </wsdl:port>
  <wsdl:port binding="tns:TCServicesSoap12" name="TCServicesSoap12">
   <soap12:address location="http://www.travel-
center.com/webservices/TCServices.asmx"/>
  </wsdl:port>
  <wsdl:port binding="tns:TCServicesHttpGet" name="TCServicesHttpGet">
   <http:address location="http://www.travel-
center.com/webservices/TCServices.asmx"/>
  </wsdl:port>
  <wsdl:port binding="tns:TCServicesHttpPost" name="TCServicesHttpPost">
   <http:address location="http://www.travel-
center.com/webservices/TCServices.asmx"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Fichier *HotelReservation*:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
 targetNamespace="http://tempuri.org/"
```

```
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:tns="http://tempuri.org/"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
 <wsdl:types>
  <s:schema elementFormDefault="qualified"
   targetNamespace="http://tempuri.org/">
   <s:element name="HelloWorld">
    <s:complexType/>
   </s:element>
   <s:element name="HelloWorldResponse">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="0" name="HelloWorldResult"
type="s:string"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:element name="GetHotelListing">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="1" name="HotelId" type="s:int"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:element name="GetHotelListingResponse">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="0" name="GetHotelListingResult"
type="tns:Hotel"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:complexType name="Hotel">
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="1" name="Id" type="s:int"/>
     <s:element maxOccurs="1" minOccurs="0" name="Name" type="s:string"/>
     <s:element maxOccurs="1" minOccurs="0" name="Address" type="s:string"/>
     <s:element maxOccurs="1" minOccurs="0" name="Phone" type="s:string"/>
    </s:sequence>
   </s:complexType>
   <s:element name="MakeReservation">
```

```
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="0" name="Resv" type="tns:Reservation"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:complexType name="Reservation">
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="1" name="Id" type="s:int"/>
     <s:element maxOccurs="1" minOccurs="1" name="CustomerId" type="s:int"/>
     <s:element maxOccurs="1" minOccurs="1" name="ArrivalDate" type="s:dateTime"/>
     <s:element maxOccurs="1" minOccurs="1" name="DepartureDate" type="s:dateTime"/>
     <s:element maxOccurs="1" minOccurs="0" name="RoomType" type="s:string"/>
     <s:element maxOccurs="1" minOccurs="1" name="RoomRate" type="s:float"/>
    </s:sequence>
   </s:complexType>
   <s:element name="MakeReservationResponse">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="1" name="MakeReservationResult"
type="s:int"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:element name="CancelReservation">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="1" name="HotelReservationId" type="s:int"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:element name="CancelReservationResponse">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="1" name="CancelReservationResult"
type="s:boolean"/>
     </s:sequence>
    </s:complexType>
   </s:element>
   <s:element name="ViewReservation">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="1" name="CustomerId" type="s:int"/>
     </s:sequence>
    </s:complexType>
   </s:element>
```

```
   <s:element name="ViewReservationResponse">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="0" name="ViewReservationResult"
type="tns:Reservation"/>
     </s:sequence>
    </s:complexType>
   </s:element>
  </s:schema>
 </wsdl:types>
 <wsdl:message name="HelloWorldSoapIn">
  <wsdl:part element="tns:HelloWorld" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="HelloWorldSoapOut">
  <wsdl:part element="tns:HelloWorldResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetHotelListingSoapIn">
  <wsdl:part element="tns:GetHotelListing" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="GetHotelListingSoapOut">
  <wsdl:part element="tns:GetHotelListingResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="MakeReservationSoapIn">
  <wsdl:part element="tns:MakeReservation" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="MakeReservationSoapOut">
  <wsdl:part element="tns:MakeReservationResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="CancelReservationSoapIn">
  <wsdl:part element="tns:CancelReservation" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="CancelReservationSoapOut">
  <wsdl:part element="tns:CancelReservationResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="ViewReservationSoapIn">
  <wsdl:part element="tns:ViewReservation" name="parameters"/>
 </wsdl:message>
 <wsdl:message name="ViewReservationSoapOut">
  <wsdl:part element="tns:ViewReservationResponse" name="parameters"/>
 </wsdl:message>
 <wsdl:portType name="HotelReservationSoap">
  <wsdl:operation name="HelloWorld">
   <wsdl:input message="tns:HelloWorldSoapIn"/>
   <wsdl:output message="tns:HelloWorldSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="GetHotelListing">
```

```
   <wsdl:input message="tns:GetHotelListingSoapIn"/>
   <wsdl:output message="tns:GetHotelListingSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="MakeReservation">
   <wsdl:input message="tns:MakeReservationSoapIn"/>
   <wsdl:output message="tns:MakeReservationSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="CancelReservation">
   <wsdl:input message="tns:CancelReservationSoapIn"/>
   <wsdl:output message="tns:CancelReservationSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="ViewReservation">
   <wsdl:input message="tns:ViewReservationSoapIn"/>
   <wsdl:output message="tns:ViewReservationSoapOut"/>
  </wsdl:operation>
 </wsdl:portType>
  <wsdl:port binding="tns:HotelReservationSoap" name="HotelReservationSoap">
   <soap:address
location="http://dotnet.borland.com/ws/Travel/HotelReservation.asmx"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Fichier *AniteTravelWS:*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
 targetNamespace="http://www.anite.com/AniteTravelWS"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:tns="http://www.anite.com/AniteTravelWS"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
 <wsdl:types>
  <s:schema elementFormDefault="qualified"
   targetNamespace="http://www.anite.com/AniteTravelWS">
   <s:element name="GetFares">
    <s:complexType>
     <s:sequence>
      <s:element maxOccurs="1" minOccurs="0" name="XMLRequestDoc">
       <s:complexType mixed="true">
```

```
      <s:sequence>
       <s:any/>
      </s:sequence>
     </s:complexType>
    </s:element>
   </s:sequence>
  </s:complexType>
 </s:element>
 <s:element name="GetFaresResponse">
  <s:complexType>
   <s:sequence>
    <s:element maxOccurs="1" minOccurs="0" name="GetFaresResult">
     <s:complexType mixed="true">
      <s:sequence>
       <s:any/>
      </s:sequence>
     </s:complexType>
    </s:element>
   </s:sequence>
  </s:complexType>
 </s:element>
 <s:element name="UserAuthentication" type="tns:UserAuthentication"/>
 <s:complexType name="UserAuthentication">
  <s:sequence>
   <s:element maxOccurs="1" minOccurs="0" name="Username" type="s:string"/>
   <s:element maxOccurs="1" minOccurs="0" name="Password" type="s:string"/>
   <s:element maxOccurs="1" minOccurs="0" name="Organization" type="s:string"/>
   <s:element maxOccurs="1" minOccurs="0" name="Domain" type="s:string"/>
  </s:sequence>
 </s:complexType>
 <s:element name="GetFaresWithAvail">
  <s:complexType>
   <s:sequence>
    <s:element maxOccurs="1" minOccurs="0" name="XMLRequestDoc">
     <s:complexType mixed="true">
      <s:sequence>
       <s:any/>
      </s:sequence>
     </s:complexType>
    </s:element>
   </s:sequence>
  </s:complexType>
 </s:element>
 <s:element name="GetFaresWithAvailResponse">
  <s:complexType>
   <s:sequence>
```

```
    <s:element maxOccurs="1" minOccurs="0" name="GetFaresWithAvailResult">
     <s:complexType mixed="true">
      <s:sequence>
       <s:any/>
      </s:sequence>
     </s:complexType>
    </s:element>
   </s:sequence>
  </s:complexType>
 </s:element>
 <s:element name="GetFareAvail">
  <s:complexType>
   <s:sequence>
    <s:element maxOccurs="1" minOccurs="0" name="XMLRequestDoc">
     <s:complexType mixed="true">
      <s:sequence>
       <s:any/>
      </s:sequence>
     </s:complexType>
    </s:element>
   </s:sequence>
  </s:complexType>
 </s:element>
 <s:element name="GetFareAvailResponse">
  <s:complexType>
   <s:sequence>
    <s:element maxOccurs="1" minOccurs="0" name="GetFareAvailResult">
     <s:complexType mixed="true">
      <s:sequence>
       <s:any/>
      </s:sequence>
     </s:complexType>
    </s:element>
   </s:sequence>
  </s:complexType>
 </s:element>
 <s:element name="BookFare">
  <s:complexType>
   <s:sequence>
    <s:element maxOccurs="1" minOccurs="0" name="XMLRequestDoc">
     <s:complexType mixed="true">
      <s:sequence>
       <s:any/>
      </s:sequence>
     </s:complexType>
    </s:element>
```

```
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="BookFareResponse">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="BookFareResult">
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="GetAccomAvail">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="XMLRequestDoc">
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="GetAccomAvailResponse">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="GetAccomAvailResult">
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="GetPriceSummary">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="XMLRequestDoc">
```

```
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="GetPriceSummaryResponse">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="GetPriceSummaryResult">
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="MakeReservation">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="XMLRequestDoc">
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
   </s:complexType>
  </s:element>
  <s:element name="MakeReservationResponse">
   <s:complexType>
    <s:sequence>
     <s:element maxOccurs="1" minOccurs="0" name="MakeReservationResult">
      <s:complexType mixed="true">
       <s:sequence>
        <s:any/>
       </s:sequence>
      </s:complexType>
     </s:element>
    </s:sequence>
```

```
    </s:complexType>
   </s:element>
  </s:schema>
</wsdl:types>
<wsdl:message name="GetFaresSoapIn">
 <wsdl:part element="tns:GetFares" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetFaresSoapOut">
 <wsdl:part element="tns:GetFaresResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetFaresUserAuthentication">
 <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
</wsdl:message>
<wsdl:message name="GetFaresWithAvailSoapIn">
 <wsdl:part element="tns:GetFaresWithAvail" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetFaresWithAvailSoapOut">
 <wsdl:part element="tns:GetFaresWithAvailResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetFaresWithAvailUserAuthentication">
 <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
</wsdl:message>
<wsdl:message name="GetFareAvailSoapIn">
 <wsdl:part element="tns:GetFareAvail" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetFareAvailSoapOut">
 <wsdl:part element="tns:GetFareAvailResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetFareAvailUserAuthentication">
 <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
</wsdl:message>
<wsdl:message name="BookFareSoapIn">
 <wsdl:part element="tns:BookFare" name="parameters"/>
</wsdl:message>
<wsdl:message name="BookFareSoapOut">
 <wsdl:part element="tns:BookFareResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="BookFareUserAuthentication">
 <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
</wsdl:message>
<wsdl:message name="GetAccomAvailSoapIn">
 <wsdl:part element="tns:GetAccomAvail" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetAccomAvailSoapOut">
 <wsdl:part element="tns:GetAccomAvailResponse" name="parameters"/>
</wsdl:message>
```

```
<wsdl:message name="GetAccomAvailUserAuthentication">
 <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
</wsdl:message>
<wsdl:message name="GetPriceSummarySoapIn">
 <wsdl:part element="tns:GetPriceSummary" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetPriceSummarySoapOut">
 <wsdl:part element="tns:GetPriceSummaryResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="GetPriceSummaryUserAuthentication">
 <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
</wsdl:message>
<wsdl:message name="MakeReservationSoapIn">
 <wsdl:part element="tns:MakeReservation" name="parameters"/>
</wsdl:message>
<wsdl:message name="MakeReservationSoapOut">
 <wsdl:part element="tns:MakeReservationResponse" name="parameters"/>
</wsdl:message>
<wsdl:message name="MakeReservationUserAuthentication">
 <wsdl:part element="tns:UserAuthentication" name="UserAuthentication"/>
</wsdl:message>
<wsdl:portType name="AniteTravelWSSoap">
 <wsdl:operation name="GetFares">
  <documentation
   xmlns="http://schemas.xmlsoap.org/wsdl/">
        Returns all available fares (without travel/journey availability)
  </documentation>
  <wsdl:input message="tns:GetFaresSoapIn"/>
  <wsdl:output message="tns:GetFaresSoapOut"/>
 </wsdl:operation>
 <wsdl:operation name="GetFaresWithAvail">
  <documentation
   xmlns="http://schemas.xmlsoap.org/wsdl/">
        Returns all available fares with travel/journey availability
  </documentation>
  <wsdl:input message="tns:GetFaresWithAvailSoapIn"/>
  <wsdl:output message="tns:GetFaresWithAvailSoapOut"/>
 </wsdl:operation>
 <wsdl:operation name="GetFareAvail">
  <documentation
   xmlns="http://schemas.xmlsoap.org/wsdl/">
        Returns travel/journey availability for a selected fare
  </documentation>
  <wsdl:input message="tns:GetFareAvailSoapIn"/>
  <wsdl:output message="tns:GetFareAvailSoapOut"/>
 </wsdl:operation>
```

```
  <wsdl:operation name="BookFare">
   <documentation
    xmlns="http://schemas.xmlsoap.org/wsdl/">
         Book/Sell selected fare
   </documentation>
   <wsdl:input message="tns:BookFareSoapIn"/>
   <wsdl:output message="tns:BookFareSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="GetAccomAvail">
   <documentation
    xmlns="http://schemas.xmlsoap.org/wsdl/">
         Get accommodation availability
   </documentation>
   <wsdl:input message="tns:GetAccomAvailSoapIn"/>
   <wsdl:output message="tns:GetAccomAvailSoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="GetPriceSummary">
   <documentation
    xmlns="http://schemas.xmlsoap.org/wsdl/">
         Returns a summary of prices for all the travel components added to the
reservation so far
   </documentation>
   <wsdl:input message="tns:GetPriceSummarySoapIn"/>
   <wsdl:output message="tns:GetPriceSummarySoapOut"/>
  </wsdl:operation>
  <wsdl:operation name="MakeReservation">
   <documentation
    xmlns="http://schemas.xmlsoap.org/wsdl/">
         Makes (Completes) a reservation and returns a locator
   </documentation>
   <wsdl:input message="tns:MakeReservationSoapIn"/>
   <wsdl:output message="tns:MakeReservationSoapOut"/>
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="AniteTravelWSSoap" type="tns:AniteTravelWSSoap">
  <wsdl:service name="AniteTravelWS">
  <wsdl:port binding="tns:AniteTravelWSSoap" name="AniteTravelWSSoap">
   <soap:address location="http://194.203.47.11/AniteTravelWS/AniteTravelWS.asmx"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Bibliographie

Agarwal, V., Arjona, L., D., Lemmens, R. 2001. « E-Performance: The Path to Rational Exuberance », The McKinsey Quarterly, n.1, 31-43.

Aleph: <http://aleph-webservices.com/>

Apple : < http://developer.apple.com/documentation/WebObjects/Web_Services/Web_Services.pdf >

Atkinson, B., Della-Libera, G., Hada, S., Hondo, M., Hallam-Baker, P., Kaler, C., Klein, J., LaMacchia, B., Leach, P., Manferdelli, J., Maruyama, H., Nadalin, A., Nagaratnam, N., Prafullchandra, H., Shewchuk, J., Simon, D. 2002. « Web Service Security, WS-Security, Version 1.0 ».

Babin, G., Leblanc, M. 2003. « Les Web Services et Leur Impact Sur Le Commerce B2B ».

Bauer, B., Huget, M.P. 2004. « Modelling Web Service Composition with UML 2.0 », *International Journal of Web Engineering and Technology*. Vol. 1, No. 4, pp. 484-501.

BPEL4WS : <http://www.ibm.com/developerworks/library/specification/ws-bpel/>, révisé le 8 février 2007.

BizTalk : Microsoft Corporation. Microsoft biztalk server. <http://www.microsoft.com/biztalk>

Bleul, S., Weise, T., Geihs, K. 2007. « Making a Fast Semantic Service Composition System Faster », The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007), pp. 517-520.

Cardoso, J., Miller, J., Su, J., Pollock, J. 2005. « Academic and Industrial Research : Do their Approaches Differ in Adding Semantics to Web Services », First International Workshop on Semantic Web Services and Web Processes Composition, Springer-Verlag Heidelberg, Vol. 3387, pp. 14-21

Cerami, E. 2002. « Web Services Essentials. Distributed Applications with XML-RPC, SOAP, UDDI & WSDL ».

Channa, N., Li, S., Shaikh, A-W., Fu, X., « Constraint Satisfaction in Dynamic Web Service Composition », *Proceedings of the sixteenth International Workshop on Database and Expert Systems Applications,* Copenhagen, Denmark, August.

Chi, Ed. H. 2000. «A Taxonomy of Visualization Techniques Using the Data State Reference Model».

Colgrave J., Akkiraju R. et Goodwin R. 2004. « External Matching in UDDI », Proceedings of IEEE International Conference on Web Services (ICWS), p.226, San Diego.

Computer Industry Almanac: Computer Industry Almanac Inc., <http://www.c-i-a.com/index.htm>

Cummings, E.M., Carr, K.S. 2001. « How Does your Garden Tank? », Darwin.

DAML+OIL : Horrocks, I., van Harmelen, F., Patel-Schneider, P.F. 2001. Reference description of the DAML+OIL ontology markup language.

Dustdar, S., Schreiner, W. 2005. « A Survey on Web Services Composition », *Int. J. Web and Grid Services, Vol.1, No.1.*

Gannon, D., Bramley, R., Fox, G., Smallen, S., Rossi, A., Ananthakrishnan, R., Bertrand, F., Chiu, K., Farrellee, M., Govindaraju, M., Krishnan, S., Ramakrishnan, L., Simmhan, Y., Slominski, A., Ma, Y., Olariu, C., Rey-Cenvaz, N., Department of Computer Science, Indiana University, « Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications ».

Domingue, J., Cabral, L., Hakimpour, F., Sell, D., Motta, E. 2004. « IRS-III : A Platform and Infrastructure for Creating WSMO-based Semantic Web Services », Proc. Of the workshop on WSMO Implementations in Germany.

Erenkrantz, J.R. 2002. « Web Services : SOAP, UDDI and Semantic Web ».

Frayret, J-M. 2002. « L'introduction des affaires électroniques dans l'industrie des produits forestiers. Une solution à la complexité du réseau de création de valeur », Faculté de foresterie et de géomatique, Université de Laval.

Friesen, A., Namiri, K. 2006. « Towards Semantic Service Selection for B2B Integration », ACM International Conference Proceeding Series; Vol. 155, Workshop proceedings of the sixth international conference on Web engineering, Palo Alto, California, Article No 17.

Garcia, E., Henriet, J. 2004. « Les services Web ».

Gerhards, E., 2006. « B2B Group Meeting. Latest Developments in the Field of B2B e-markets ».

Gottschalk, K., Graham, S., Kreger, K., Snell, J. 2002. « Introduction to Web Services Architecture », IBM Systems Journal, Vol 41, No2.

Grigoryan, A.A., 2006. « B2B E-commerce in the United States, Europe and Japan: A Comparative Study », Journal of Applied Sciences, Asian Network for Scientific Information.

Hamadi, R., Benatallah, B. 2003. A Petri Net-Based Model for Web Service Composition, *Proceedings of the Fourteenth Australasian database conference on Database technologies 2003*, ACM press, pp. 191 – 200.

Hansen, C., Fresco, F., Malcherek, P. 2002. « Le principe du service web ».

He, M., Jennings, N. R., Leung, H. 2003. « On Agent-Mediated Electronic Commerce. IEEE Transactions on Knowledge and Data Engineering », Vol. 15, No. 4, pp. 985-1003.

Jeng, D.M. 1997. « A Petri Net Synthesis Theory for Modeling Flexible Manufacturing Systems », IEEE Transactions On Systems, Man, And Cybernetics – Part B : Cybernetics, Vol. 27, No. 2.

Kemmler, K., Kubicova, M., Musselwhite, R., Prezeau, R. 2001. « E-Performance II: The Good, the Bad and the Merely Average », The McKinsey Quarterly, n.3.

Lamport, L., <http://lamport.org/> et <http://fr.wikipedia.org/wiki/Leslie_Lamport>

Mili, H., Marcotte, O., Kabbaj, A. 1994. « Intelligent Component Retrieval for Software Reuse », in *Proceedings of the Third Maghrebian Conference on Artificial Intelligence and Software Engineering*, Rabat, Morocco, pp. 101-114.

Mili, H., Marcotte, O., Caillot, A-E., Tremblay, G. 2004. « Automatic Discovery of Web Service Compositions Using Function Covering », Technical Report, LATECE.

Morin, C. 2002. Université de Rennes 1, « Systèmes Distribués », Projet INRIA Paris.

Nicolle, C. 2002. « Définition des services web ».

North, K., « IBM and MICROSOFT Discontinue Public UDDI Registry ».

O'Sullivan, J., Edmond, D., ter Hofstede, A. 2002. « What's in a service », Distributed and Parallel Databases, 12 (2-3) : 117-133.

OWL : Dean, M. et al. 2002. OWL Web Ontology Language 1.0 reference.

OWL-S : 1.0 Release <http://www.daml.org/services/owl-s/1.0/>

OWL-S: Martin. D., , Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K. 2004. « Bringing Semantics to Web Services: The OWL-S Approach », *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, San Diego, California, USA.

Paolucci, M., Srinivasan, N., Sycara, K., Nishimura, T. 2003. « Towards a Semantic Choreography of Web Services: from WSDL to DAML-S », Proceedings of first International Conference on Web Services, ICWS '03, Las Vegas.

Poulin, J. S., Yglesias, K.P. 1993. « Experiences with a Faceted Classification Scheme in a Large Reusable Software Library (RSL) », International Business Machines Corporation.

Prefuse : <http://prefuse.org/doc/manual/> et <http://prefuse.org/doc/manual/introduction/structure/#fn1-src>

Qi, L., Lingling, G.,Yong, T., Feng, H. 2004. «An Integrated Geospatial Metadata Storing Architecture».

Rayport, J-F., Jaworski, B. 2001. e-commerce, McGraw-Hill/Irwin, Boston.

Salton G., McGill, M. 1983. « Introduction to Modern Information Retrieval », New York, McGraw-Hill.

Skogan, D., Gronmo, R., Solheim., I. 2004. « Web Service Composition in UML », *Proceedings of the 8th International IEEE Enterprise Distributed Object Computing Conference* (*EDOC*), Monterey, California..

Srivastava, B., Koehler, J. 2003. « Web Service Composition - Current Solutions and Open Problems », *ICAPS 2003 Workshop on Planning for Web Services*, p 28-35.

Sun, H., Wang, X., Zhou, B. and Zou, P. 2003. «Research and Implementation of Dynamic Web

Services Composition », APPT 2003, LNCS 2834, Springer-Verlag Berlin Heidelberg, pp.457–466.

Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N. 2003. « Automated Discovery, Interaction and Composition of Semantic Web Services », Journal of Web Semantics, Volume 1, Number 1.

Ter Beek, M.H., Bucchiarone, A., Gnesi, S. 2006. « A Survey on Service Composition Approaches: From Industrial Standards to Formal Methods ».

UDDI : <http://www.uddi.org/pubs/lru_UDDI_Technical_White_Paper.pdf>, Septembre 2000.

Verissimo, P., Rodrigues, L. 2001. « Distributed Systems for System Architects ».

W3C : « Simple Object Access Protocol (SOAP) », Version 1.1, W3C Note, <http://www.w3.org/TR/2000/NOTE-SOAP-20000508>, 15 février 2000.

W3schools : <http://www.w3schools.com/soap/>

WebSphere : IBM corporation. Websphere software, <http://www.ibm.com/software/info1/websphere>.

Wohed, P., Aalst, M.P., Dumas, M., Hofstede, A. 2002. « Pattern Based Analysis of BPEL4WS ».

Wolter, R. 2001. « Principes de base des Web Services », Microsoft corporation.

WSC : <http://www.ws-challenge.org/>

WSDL4J : <http://www.hta-bi.bfh.ch/Projects/fancyws/generated/html/x907.html>.

Zhang, W., Zhang, X. 2006. « Modeling Service Interactions Using Kahn Process Network ».