Un canevas pour l'adaptation et la substitution de services Web

Yehia Taher

N° d'ordre **125- 2009** Année 2009

THESE DE L'UNIVERSITE DE LYON

Délivrée par

L'UNIVERSITE CLAUDE BERNARD LYON 1

ECOLE DOCTORALE
Infomaths

DIPLOME DE DOCTORAT
(arrêté du 7 août 2006)

soutenue publiquement le (24 Juillet 2009)

par

Yehia TAHER

TITRE :

Un canevas pour l'adaptation et la substitution de services Web

Directeurs de thèse :

Marie-Christine Fauvet	Prof., Université Joseph Fourier, Grenoble
Djamal Benslimane	Prof., Université de Lyon 1

JURY :

Rapporteurs :	Yamine Aït-Ameur	Prof., ENSMA – Poitiers, LISI
	Michel Léonard	Prof., Université de Genève, MATIS
Examinateurs :	Olivier Boissier	Prof., Ecole des mines de Saint-Etienne, G2I
	Mohand Boughanem	Prof., Université Paul Sabatier, IRIT
Directeurs :	Marie-Christine Fauvet	Prof., Université Joseph Fourier, LIG
	Djamal Benslimane	Prof., Université de Lyon 1, LIRIS

Remerciements

Ce travail a été réalisé au sein des Laboratoires LIRIS[1] de l'université Claude Bernard Lyon 1 et LIG[2] de l'université Joseph Fourier, sous la direction de Djamal Benslimane, Professeur à l'université Claude Bernard Lyon 1, et de Marie-Christine Fauvet, Professeur à l'université Joseph Fourier, auxquels j'exprime ma gratitude pour l'encadrement de mon travail et leurs apports tant au niveau des connaissances qu'au niveau humain, mais également leurs encouragements, ainsi que leur soutiens et leur tolérance tout au long de la thèse.

Je remercie tout particulièrement Mr. Marlon Dumas de m'avoir accueilli pendant six mois dans son équipe de recherche à l'université de Queensland en Australie, mais aussi pour sa curiosité et ses conseils constructifs.

Je tiens à remercier Yamine Aït-Ameur, Professeur à l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique et, Michel Léonard, Professeur à l'université de Genève de m'avoir fait l'honneur de rapporter ce mémoire.

Je tiens à remercier également les membres du jury, Olivier Boissier, Professeur à l'Ecole des mines de Saint-Etienne, ainsi que Mohand Boughanem, Professeur à l'université Paul Sabatier.

Je tiens à remercier, enfin, ma chère famille, et mes ami(e)s qui m'ont aidé tout au long de la thèse et encouragé à aller dans cette direction.

[1] Laboratoire d'InfoRmatique en Image et Systèmes d'information
[2] Laboratoire d'Informatique de Grenoble

Table des matières

Partie I: Contexte de recherche et état d'art

Chapitre 1
Contexte et problématique

Chapitre 2
Etat de l'art

Partie II: Proposition d'un canevas pour l'adaptation d'interactions des services par génération semi-automatique d'adaptateurs

Chapitre 3
Modélisation d'interfaces de services

Chapitre 4
Adaptation d'interactions de services

Chapitre 5
Mise en œuvre avec la technologie CEP

Table des figures

Résumé

Les services Web ont émergé comme un support de développement et d'intégration d'applications ou de systèmes d'information. Dans ce cadre, les interactions entre deux applications consommateur et fournisseur, encapsulées par des services Web se font par échanges de messages. Ces échanges s'appuient sur la notion d'interfaces, qui décrivent les interactions dans lesquelles un service peut s'engager et les dépendances entre ces interactions. Dans le Web actuel, il arrive très fréquemment que de nombreux services répondent à un même ensemble de besoins fonctionnels. Ces services sont souvent offerts par le biais d'interfaces différentes.

Des nombreuses raisons, telles que la panne du service fournisseur, peuvent amener un consommateur à substituer son fournisseur habituel par un autre fournisseur qui offre la même fonctionnalité. Cette substitution provoque des incompatibilités entre l'interface du service consommateur et celle du service fournisseur substitut. Cela est dû au fait que le service consommateur n'a pas été fait en fonction de ce nouveau service fournisseur.

Les recherches que nous menons dans cette thèse visent à résoudre le problème des incompatibilités dans des interactions entre deux services consommateur et fournisseur substitut. En particulier, notre contribution s'étend, tant sur le plan théorique que sur le plan pratique. Il s'agit d'une part d'un canevas pour la génération automatique des adaptateurs des interactions entre deux services. D'autre part, nous proposons une architecture logicielle multicouche fournissant un cadre permettant une substitution transparente et flexible d'un service fournisseur par un autre service vis-à-vis du consommateur du premier.

Dans notre canevas pour la génération automatique des adaptateurs, une modélisation des interfaces de services en des automates a été adoptée. Puis, une étape de détection des incompatibilités entre ceux-ci est réalisée. Ensuite, un adaptateur des interactions entre les deux services est généré automatiquement sur la base d'incompatibilités détectées. La génération de l'adaptateur est guidée par le modèle d'automates. Cela permet de modéliser l'adaptateur indépendamment de son implémentation cible, permettant ainsi une bonne réutilisation des modèles. Une fois généré, l'automate de l'adaptateur suffisamment détaillé est projeté sur la technologie CEP *(Complexe Event Processing)*. Cette projection est réalisée à l'aide des composants cartouches (en anglais : *Templates*) que nous avons mis en œuvre. Chaque cartouche étant conçue pour générer du code exécutable (en termes *de requêtes continues*) pour la technologie CEP.

Notre architecture proposée pour la substitution de services Web s'intègre d'une part la notion de communauté de services, et d'autre part un progiciel nommé OSC (*Open Service Connectivity*). Une communauté de services est perçue comme un moyen d'exposer des descriptions communes d'un caractère fonctionnel désiré sans explicitement se référer à un service spécifique. L'OSC est un composant logiciel dont l'objectif est d'appliquer le principe de pilotes *ODBC* et *JDBC* dans un environnement à base de services Web. Plus précisément, il est responsable de gérer les interactions entre les consommateurs des communautés d'une part et les communautés d'autre part, en fournissant des fonctions permettant la sélection et la substitution de services.

Mots clés : Services Web, interface, substitution, incompatibilités structurelles et comportementales, détection, résolution, adaptateurs, automate, CEP, communauté de services, OSC.

Abstract

Web services have emerged as a support for development and integration of applications and information systems. In this context, the interactions between two consumer and supplier applications, encapsulated by Web services are done by exchanging messages. These exchanges are based on the concept of interfaces, which describe the interactions in which a service can handle and dependencies between these interactions. In the current Web, it is very often that many services meet the same set of functional requirements. These services are often delivered through different interfaces.

For many reasons, such as failure of the service provider, the consumer has to replace his usual supplier with another supplier that offers the same functionality. This substitution leads to incompatibilities between the interfaces of the service consumer and new service provider. This is because the customer service was not been developed according to the new service provider.

The research we conduct in this thesis aims to solve the problem of incompatibility in the interaction between two services; consumer and new provider. In particular, our contribution extends both in theory and in practice. It is a part of a framework for the automatic generation of adapters for interactions between two services. On the other hand, we propose a multi-layer software architecture providing a framework for transparent and flexible substitution of a service provider by another with respect to an existed consumer.

In our framework for automatic generation of adapters, services interfaces modeling using automata has been adopted. Then, a step of detecting incompatibilities between them is achieved. After that, an adapter of the interactions between the two services is generated automatically based on the detected incompatibilities. The generation of the adapter is based on the automata model. The generated adapter automaton contains a sufficient detail for the projected technology CEP (Complex Event Processing). This projection is performed using components templates that are implemented by us. Each template is designed to generate executable code (in terms of continuous requests) for the CEP technology.

Our proposed architecture for the substitution of integrated Web services is an approach for deploying communities of Web services. A community promotes the dynamic binding of Web services through software named OSC (Open Service Connectivity). A community service is seen as a way to expose common descriptions of a desired functionality without explicitly referring to a specific service. OSC is a software component whose objective is to apply the principle of ODBC and JDBC in an environment based on Web services. Specifically, it is responsible for managing interactions between communities of consumers on the one hand and communities on the other hand, providing functions for the selection and substitution of services.

Keywords: Web, interface, substitution, structural and behavioral incompatibilities, detection, resolution, adapters, automata, CEP, community, OSC.

Première partie

Contexte de la recherche et état de l’art

CHAPITRE 1

Contexte et problématique

Sommaire

1.1 Introduction

Pendant très longtemps, le dialogue entre machines nécessitait une connaissance approfondie des protocoles réseau et parfois même du matériel réseau. La recherche sur les langages de programmation et les technologies de l'information a permis le développement des architectures logicielles à base des composants distribués, et cela en fournissant des bibliothèques pour faire communiquer par des messages les composants répartis sur des machines différentes. L'échange des messages s'appuyait sur des technologies comme *CORBA*[3]et *RMI*[4]. Avec la démocratisation des réseaux et l'arrivée de l'*Internet,* les architectures à base de composants distribués ont évolué vers des architectures à base de services.

L'objectif d'une architecture à base de services *(Service-Oriented Architecture, SOA)* est de décomposer une fonctionnalité en un ensemble des fonctions basiques, appelées *services*.

[3] Common Object Request Broker Architecture
[4] Remote Method Invocation

Le terme *service* est introduit par analogie avec le monde réel : lorsqu'il utilise un service de distribution de billets de banque via un distributeur automatique, le consommateur de ce service n'a pas à connaître le fonctionnement technique du distributeur et encore moins le détail de ses connexions avec les serveurs de différentes banques impliquées. Cette simplicité permet à n'importe quel consommateur de réutiliser le service avec un minimum d'effort. Un service vise donc à être simple d'emploi et réutilisable.

Un service Web, au sens *SOA*, met à disposition de consommateurs un regroupement d'une ou de plusieurs fonctions *(opérations)* ayant un sens sur le plan métier. Le consommateur n'a pas à se préoccuper de la façon dont ces fonctions sont implantées et *a fortiori* des technologies sous-jacentes. Un service Web peut être consommé soit par un utilisateur humain via une application cliente, soit par un processus métier, ou bien un autre service Web. Dans la suite de cette thèse, les termes *consommateur* et *service consommateur* sont utilisés indifféremment et rapportent à l'un de ces trois éléments.

Les interactions entre un service et son consommateur se font par échanges de messages. Ces interactions s'appuient sur la notion d'interfaces, qui décrivent les interactions dans lesquelles un service peut s'engager et les dépendances entre ces interactions. Dans une interface, les interactions sont décrites du point de vue du service concerné sous forme d'actions *observables*, c'est-à-dire des actions d'envoi et de réception des messages encodés en *XML*.

La mise en œuvre de services Web s'appuie aujourd'hui sur l'utilisation des standards basés sur *XML*, tels que *SOAP*[5], *WSDL*[6], *BPEL*[7], *WSCI*[8], etc. *SOAP* est un protocole léger qui définit une interface de communication standard entre les services. Sa spécification définit la façon dont les messages sont échangés et dont les requêtes, réponses et contenus de messages sont encodés. Les messages échangés via *SOAP* sont en réalité des flux *XML* qui sont en général transmis au dessus *d'HTTP*. *WSDL* est un langage permettant de décrire les aspects structurels d'un service, c'est-à-dire le format de messages requis pour communiquer avec ce service, les méthodes que le client peut invoquer ainsi que la localisation du service. Quant à eux, *BPEL* et *WSCI* permettent de décrire les aspects comportementaux d'un service.

Lorsque la fonctionnalité requise par un consommateur n'est pas satisfaite par un service Web, il est possible d'en composer plusieurs pour satisfaire cette fonctionnalité requise. La mise en place d'une composition de services Web pose les problèmes tels que la sélection des services participants, et la mise en œuvre de schémas d'interactions entre ceux-ci.

[5] Simple Object Access Protocol, http://www.w3.org/TR/soap/
[6] Web Service Description language, http://www.w3.org/TR/wsdl/
[7] Business Process Execution language, http://bpel.xml.org/
[8] Web Service Choreography Interface, http://www.w3.org/TR/wsci/

L’interface d’un consommateur est généralement faite en fonction de l’interface fournie par le service fournisseur avec lequel il souhaite interagir. Lorsque le consommateur substitute son service fournisseur habituel par un autre, des incompatibilités peuvent avoir lieu entre son interface et celle du fournisseur substitut. Ces incompatibilités sont dues au fait que le consommateur n’a pas été conçu pour interagir avec ce nouveau fournisseur. Pour remédier ce problème, des mécanismes d’adaptation par détection et résolution des incompatibilités doivent être mis en œuvre. C’est sur cette problématique que nous nous concentrons dans la suite de cette thèse.

Le présent chapitre est organisé comme suit. La section 1.2 présente les concepts fondamentaux liés à la notion de service. La section 1.3 présente à travers une illustration de substitution de services la problématique liée aux incompatibilités entre les interfaces de services, ainsi que les objectifs de la thèse. La section 1.4 présente les deux contributions principales de la thèse : un canevas pour la génération automatique des adaptateurs et, une architecture logicielle facilitant la substitution de services. Enfin, le plan de la thèse est discuté dans la section 1.5

1.2 Concepts fondamentaux de la notion de service

Dans cette section, nous introduisons les différents concepts liés aux services Web. La section 1.2.1 discute du concept d’interfaces de service et, expose le principe d’interactions par échanges de messages entre services. La section 1.2.3 présente le principe de composition de services Web. La compatibilité des interfaces des services est définie dans la section 1.2.4.

1.2.1 Interfaces et échanges de messages

Pour utiliser un service, le consommateur émet une requête vers le fournisseur. Celui-ci renvoie (en général) une réponse. A chaque requête envoyée par le consommateur correspond une opération de réception effectuée par le service et, à chaque réponse renvoyée par le service correspond une opération d’envoi effectuée par celui-ci.

L’interface structurelle spécifie la liste des opérations offertes par le service et, pour chaque opération les messages de requête et de réponse échangés avec les consommateurs du service. Le triplet *[nom de l’opération, message de requête, message de réponse]* est appelé signature de l’opération. Les messages transportent des informations métiers, dites sérialisées. Pour cela l’interface doit aussi préciser la structure et le format de chaque information. Les opérations sont accessibles via un ou plusieurs protocoles de communications *(HTTP, FTP, SMTP, etc.)*. Enfin, le service est localisé sur une ressource physique, autrement dit un

serveur. Le protocole et la localisation du service sont dépendants d'une technologie réseau donnée. Alors que, les opérations et les signatures sont indépendantes de toute technologie. *WSDL* est le langage standard utilisé pour décrire l'interface structurelle de services Web.

La figure 1.1 illustre les composants essentiels d'une interface structurelle. Une Interface de service reçoit un Message qui contient des Informations métiers. Le transport du message est effectué par le biais d'un Protocole de transport qui utilise l'Adresse de service.

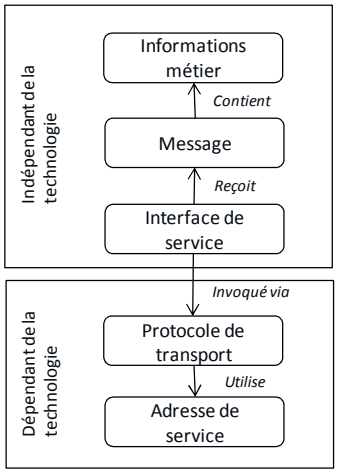

Figure 1.1 La description de l'interface structurelle

L'interface comportementale permet de spécifier la séquence d'invocations des opérations décrites dans l'interface structurelle d'un service. Différents langages ont été introduits récemment pour décrire l'interface comportementale de services, notamment : *BPEL*, *WSCI*, *BPMN*.

Les interactions entre consommateurs et fournisseurs sont effectuées par échanges de messages. Le consommateur doit respecter les interfaces d'un service, c'est-à-dire se conformer à un protocole de communication et à échanger des messages d'envoi et/ou réception selon la structure et la séquence d'invocations définies par les interfaces. Un consommateur envoie un message vers le service fournisseur, celui-ci reçoit le message et répond avec un autre qui sera reçu par le consommateur. Ainsi, lors de l'interaction entre deux services, leurs rôles d'envoi et de réception de messages sont inversés en permanence. Quand le consommateur est sur le point d'envoyer un message, le fournisseur est sur le point de recevoir ce message. Quand le fournisseur s'apprête à envoyer un message, le consommateur s'apprête à recevoir ce message.

D'un point de vue technique, le format de messages est fixé par le standard *SOAP* basé sur *XML*. Les messages *SOAP* sont transportés sur le réseau en s'appuyant sur différentes

protocoles standards de communication tels que HTTP, FTP, Java RMI, .NET Remoting, SMTP, etc.

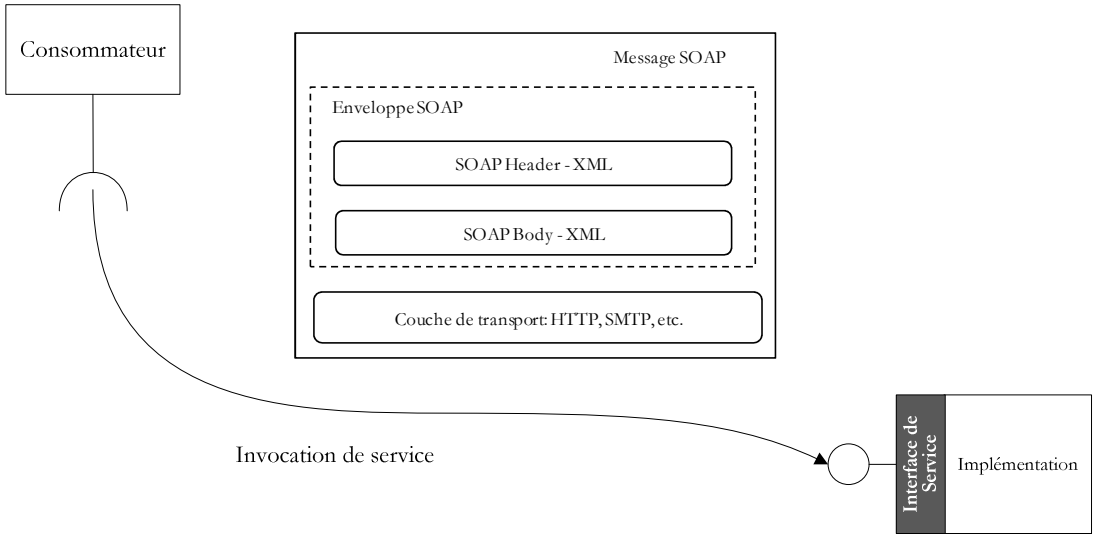

Figure 1.2 Fonctionnement de SOAP

Comme l'illustre la figure 1.2, un message *SOAP* est construit par un consommateur dans la perspective d'être envoyé à destination du service fournisseur concerné. Le message est composé de deux parties :

- La partie Header porte des informations complémentaires pour le traitement des données (identification de l'émetteur du message, règle de sécurité pour la lecture du message, algorithme à utiliser pour la lecture de message, etc.)

- La partie Body porte les données propres au message, et matérialise la requête ou la réponse *SOAP* (structure de données spécifiques).

Pour illustrer un cas concret des interactions entre services, nous considérons dans la figure 1.3 un exemple inspiré du service Web *Amazon*[9] pour la vente des produits en ligne. L'interface structurelle décrit les différentes opérations offertes par le service, notamment : l'opération de commande d'un article, validation de panier, envoi de devis, annulation de commande, paiement, et livraison. La séquence d'exécution de ces opérations est décrite dans l'interface comportementale. Par exemple, l'opération commande d'un article suivie soit de l'exécution de l'opération elle-même pour commander un deuxième article, ou bien de l'opération validation de panier pour lancer le calcul de devis de la commande. Ainsi, l'opération validation de panier suivie de l'opération envoi de devis. Celle-ci est suivie soit de

[9] http://www.amazon.fr/

l’opération annulation ou bien de l’opération paiement suivie à son tour de l’opération livraison.

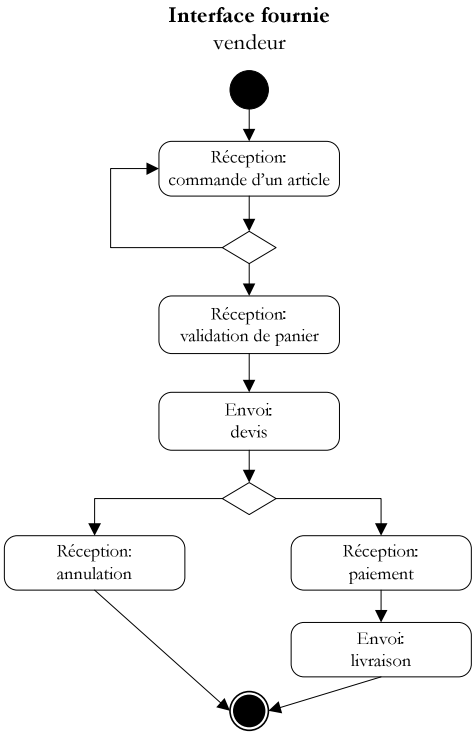

Figure 1.3 Interface du service vendeur

Les interactions entre le consommateur et le service sont initiées par le premier qui doit être en mesure d’envoyer un ou plusieurs messages pour commander respectivement un ou plusieurs articles, puis d’envoyer un message pour valider son panier, de recevoir un message contenant le devis de sa commande et, en fonction de ce devis soit envoyer un message pour annuler sa commande, soit envoyer un message pour effectuer le paiement de la commande, enfin, recevoir un message qui confirme la livraison de sa commande.

Pour être en mesure de procéder de cette manière, il est requis qu’un consommateur soit muni d’une interface lui permettant l’envoi et la réception des messages de manière adaptée à la structure et la séquence d’envoi/réception de messages telle qu’elles sont définies dans les interfaces structurelle et comportementale du service.

La figure 1.4 illustre l’interface du service consommateur, requise pour réussir les interactions avec le service fournisseur. En effet, à chaque action de réception de message dans l’interface du service fournisseur correspond une action d’envoi de message dans l’interface requise par le service consommateur et, la structure du message associé à l’action d’envoi est conforme à la structure du message associé à l’action de réception.

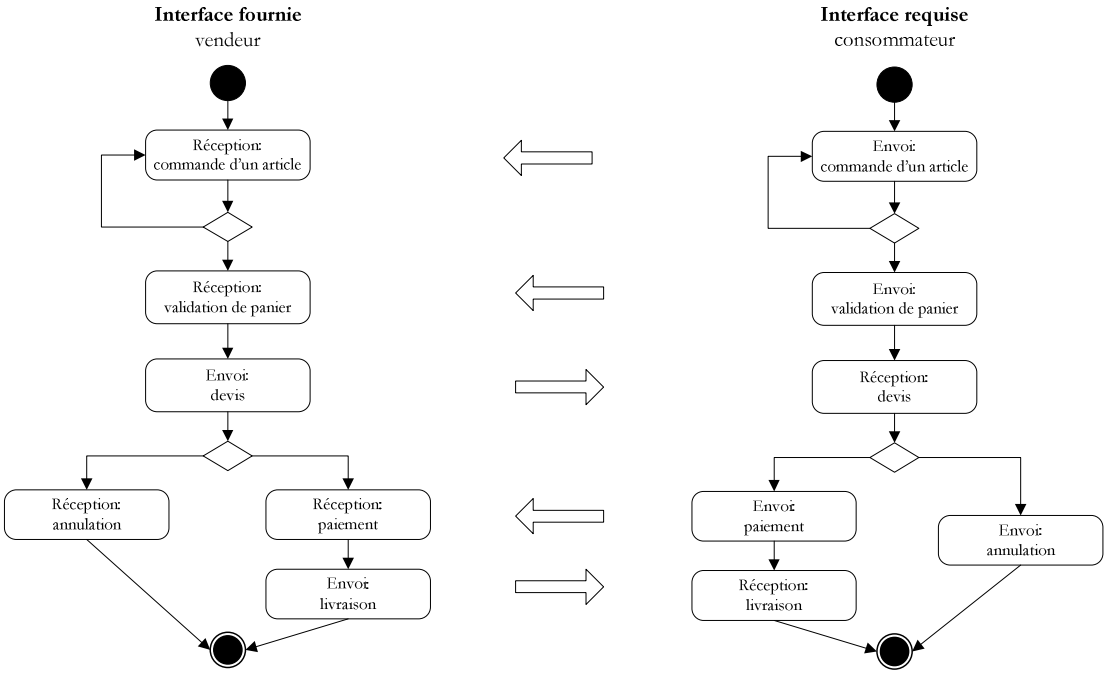

Figure 1.4 Interface fournie *vs* interface requise

Dans la suite de ce document de thèse, le terme *interface requise* est utilisé pour référer à l'interface que le service consommateur doit fournir dans son interaction avec un service fournisseur. L'interface de ce dernier est parfois nommée *interface fournie*.

1.2.2 Composition de services Web

L'architecture à base de services favorise la construction de nouveaux services par composition de services existants. Benatallah et Coll. [BDDM05], définissent une composition de services Web comme étant un moyen efficace pour créer, exécuter, et maintenir des services qui dépendent d'autres services.

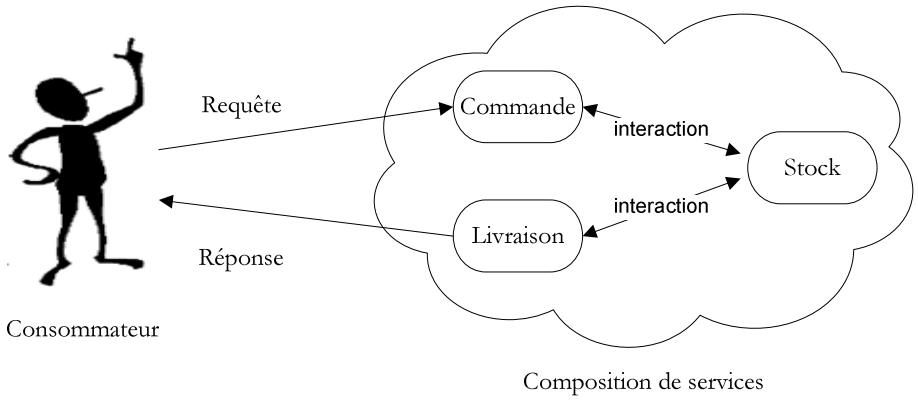

Figure 1.5 Vue générale d'une composition de services

La figure 1.5 illustre une vue générale d'une composition de services Web. Le consommateur (logiciel ou humain) établit une requête satisfaite par l'exécution des trois services Web Commande, Stock, et Livraison. La requête de l'utilisateur est transmise au premier service Commande. Celui-ci passe la commande au service Stock qui à son tour interagit avec le service Livraison pour envoyer la commande au consommateur.

Deux types de composition on été identifiés par les différentes approches qui portent sur la composition de services [ACKM04, BDDM05, CAH05], à savoir : l'orchestration et la chorégraphie.

L'orchestration est définie comme un ensemble d'actions à réaliser par l'intermédiaire de services Web. La logique de l'enchaînement de services (séquentiel, parallèle, etc.) est décrite, comme dans le flot de tâches (en anglais : *workflow*) classique, par un modèle d'orchestration directement exécuté par un service Web jouant le rôle du *chef d'orchestre*, appelé moteur d'exécution du processus. L'orchestration décrit également la logique de contrôle et le flux de données internes entre le moteur d'exécution et les services. Ces interactions correspondent aux appels, effectués par le moteur, d'actions proposées par les services composants.

La chorégraphie, ou bien la composition interactive de services, permet de décrire une composition comme étant un moyen d'atteindre un but commun en s'appuyant sur un ensemble de services. Les interactions entre les services faisant partie de la composition sont décrites par des flots de contrôle.

A l'inverse des orchestrations, dans une chorégraphie il n'y a pas de coordinateur central, la coordination est distribuée dans des pairs.

1.2.3 Compatibilité d'interfaces

Comme nous l'avons illustré auparavant, l'interface requise d'un consommateur doit respecter les interfaces d'un service aussi bien sur le plan structurel que sur le plan comportemental. Sinon, les interactions ne peuvent pas aboutir entre les deux services.

Une interface requise I_R est dite *compatible* à une interface fournie I_F sur le plan structurel si et seulement si la structure de chaque message envoyé (respectivement reçu) décrite dans l'interface requise correspond à la structure du message reçu (respectivement envoyé) décrite dans l'interface fournie.

Une interface requise I_R est dite *compatible* à une interface fournie I_F sur le plan comportementale si et seulement si la séquence des opérations d'envoi (respectivement de

réception) décrite dans l'interface requise est cohérente avec la séquence des opérations de réception (respectivement d'envoi) décrite dans l'interface fournie.

Une interface requise I_R est dite *compatible* à une interface fournie I_F si et seulement elle est *compatible* à I_F sur le plan structurel ainsi que sur le plan comportemental.

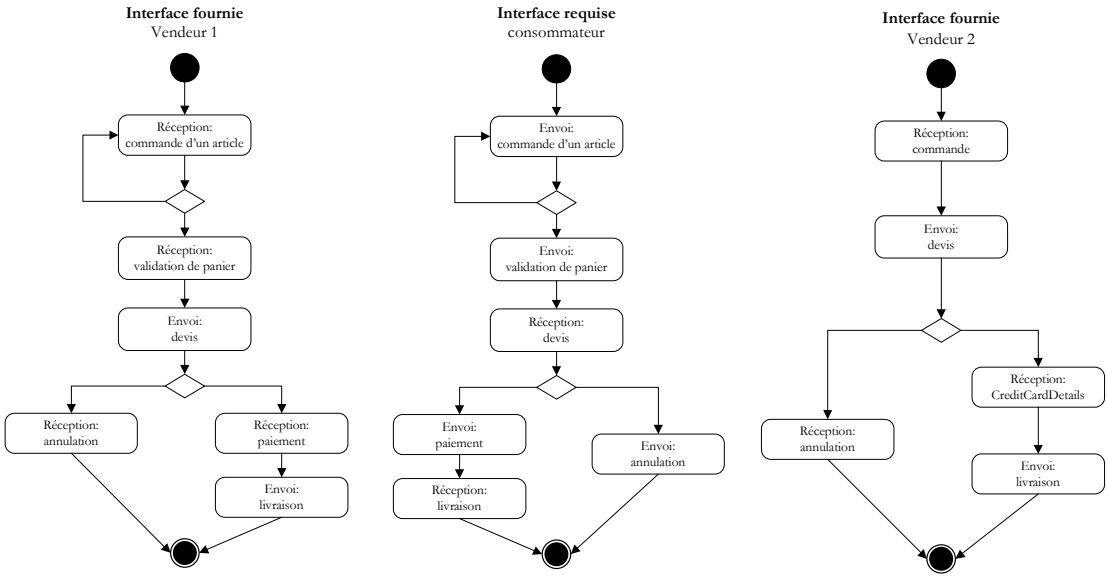

Figure 1.6 compatibilités des interfaces fournie et requise

Dans la figure 1.6, l'interface requise par le consommateur est *compatible* à l'interface fournie par le Vendeur 1 car pour chaque opération d'envoi (respectivement de réception) dans la première interface correspond une opération de réception (respectivement d'envoi) dans la deuxième interface et, les structures de ces messages correspondent (en admettant que dans cet exemple les messages de même nom ont la même structure). Alors que, cette interface requise n'est pas *compatible* à l'interface fournie par le Vendeur 2, ni sur le plan structurel, ni sur le plan comportemental. Du point de vue structurel, les deux actions Envoi : commande d'un article et Réception : commande n'ont pas de structures correspondantes. Du point de vue comportemental, l'action Envoi : commande d'un article est exécuté plusieurs fois dans l'interface requise, alors que l'action Réception : commande est exécutée une fois unique.

1.3 Problématique et objectifs

Dans cette section, nous détaillons la problématique liée aux incompatibilités lors des interactions entre des services Web. Notamment, la section 1.3.1 justifie l'étude des

incompatibilités entre services à travers une illustration de substitution de services Web. La section 1.3.2 définit le problème de recherche traité dans cette thèse en identifiant la dimension des incompatibilités concernées. Les objectifs de la thèse sont exposés dans la section 1.3.3.

1.3.1 Illustration : substitution de services Web

Plusieurs situations peuvent justifier l'étude des incompatibilités entre l'interface d'un service consommateur et celle du service fournisseur avec lequel il souhaite interagir. Dans cette section, nous mettons l'accent sur l'une parmi ces situations, à savoir la substitution de services Web.

Dans le Web actuel, il arrive très fréquemment que de nombreux services répondent à un même ensemble de besoins fonctionnels : pour réaliser l'achat en ligne d'un certain produit, un client peut mettre en concurrence plusieurs compagnies de vente. Ces services sont offerts par le biais des interfaces différentes.

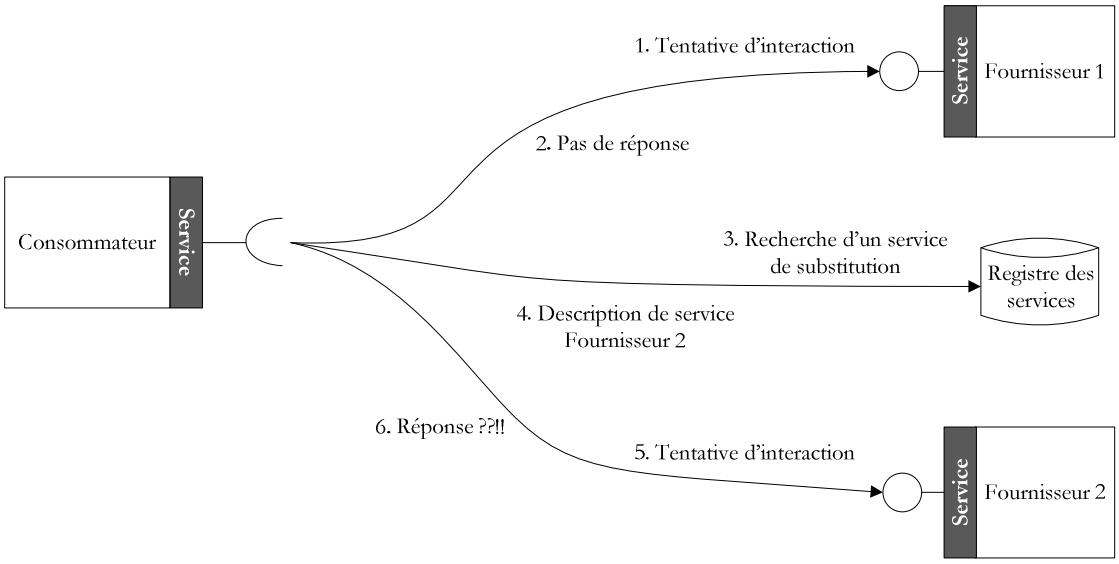

Figure 1.7 Exemple de substitution de services Web

Dans l'exemple de la figure 1.7, le service consommateur tente d'interagir (1. Tentative d'interaction) avec son service fournisseur habituel (Fournisseur 1). Étant donné que ce dernier ne répond pas (2. Pas de réponse), le service consommateur va en chercher un autre dans un registre des services (3. Recherche d'un service de substitution). Lors de la sélection du service, les fonctionnalités offertes doivent être correctement identifiées afin de répondre aux exigences du consommateur. Une fois le service trouvé (4. Description de service Fournisseur 2), le client se connecte à celui-ci et tente d'interagir avec (5. Tentative

d'interaction). Lors des interactions entre les deux services, les messages échangés doivent être adaptés à l'interface du service auquel ils sont destinés. Sinon, les interactions ne peuvent pas aboutir (6. Réponse ??!!).

1.3.2 Problèmes soulevés

Deux problèmes principaux découlent de la substitution de services Web :

- La sélection dynamique du service substitut par le consommateur au moment de son exécution: domaine de recherche très vaste.

- Les incompatibilités entre l'interface du service consommateur et celle du service substitut, dues au fait que le consommateur n'a pas été fait en fonction de ce service fournisseur.

Les recherches que nous menons dans cette thèse portent principalement sur l'étude du problème des incompatibilités entre deux interfaces. Etudier les incompatibilités entre deux interfaces permet de mettre en exergue leurs différences qui peuvent générer des incompatibilités dans les interactions de services. Le but de cette étude est de mettre en place les adaptations nécessaires pour résoudre les incompatibilités lors des interactions entre deux services.

Une approche triviale pour traiter les incompatibilités entre deux interfaces consiste à propager des modifications sur la première interface de manière à devenir compatible avec la deuxième. Mais, cette modification amène à modifier aussi les autres partenaires du service qui peuvent être à leur tour utilisés par d'autres services, ce qui nécessite également la modification de ces services, et ainsi de suite. Cette approche est très coûteuse en termes d'effort et de temps car il s'agit de modifier tous les partenaires du service modifié ainsi que les partenaires de ceux-ci.

Une autre approche alternative consiste à mettre en place un composant spécifique dénommé *adaptateur* [BCG+05]. Celui-ci, placé en amont du service consommateur, permet de réconcilier les interactions entre les deux services par détection et résolution des incompatibilités.

Des incompatibilités de différentes classes sont traitées par l'adaptateur, à savoir : structurelles, comportementales, et sémantiques. En particulier, Mrissa [MRI07] propose dans sa thèse un mécanisme de médiation - assuré par un composant appelé médiateur - capable d'adapter les données transmises par les services en tenant compte des différences de représentation structurelle et sémantique de ces données. La solution proposée pour

l'adaptation selon leur dimension structurelle s'appuie sur des mécanismes de mise en correspondance de messages de types différents. Celle proposée pour l'adaptation selon la dimension sémantique s'appuie sur la génération d'automates capables d'interpréter correctement les données décrites par l'interface du service. Pour ce faire, un enrichissement des descriptions d'interface doit être proposé afin de rendre le contexte sémantique des données explicite et accessible aux machines. De plus, des mécanismes à base de règles ont été développés afin de pouvoir adapter des données hétérogènes mais sémantiquement équivalentes.

Dans sa thèse [AB08], Aït-Bachir propose un mécanisme de médiation qui porte sur les aspects comportementaux des services. Plus précisément, il propose de réconcilier les conversations entre un service et une application cliente qui ne peuvent pas aboutir car l'interface fournie par le service a été modifiée, et n'est plus compatible avec celle attendue par le consommateur. Dans ce travail, la réconciliation des conversations s'appuie sur un ensemble limité de cas identifiés à partir des évolutions possibles d'une interface, à savoir l'ajout, la suppression, et la modification d'opérations.

Alors qu'une attention particulière à été donnée aux incompatibilités structurelles et comportementales, des incompatibilités mixtes (comportementales avec des propriétés structurelles) ont été peu considérées [DSW06, BCG+05].

Dans notre étude l'accent est mis sur les incompatibilités mixtes, qui touchent à la fois aux aspects structurels et comportementaux des interfaces des services ainsi que nous le détaillons ci-après.

Des multiples exécutions d'une opération *versus* une exécution unique d'une autre opération

Dans la figure 1.8, l'Interface fournie du service Vendeur 1 décrit le comportement requis par un consommateur pour placer une commande. Les lignes de la commande doivent être envoyées séparément en sollicitant n fois *(n étant le nombre des lignes de commande)* l'opération Réception : ligne de commande, suivi de l'exécution de l'opération Réception : validation de Panier pour annoncer la complétude de la commande. L'interface du consommateur est compatible à cette interface.

L'Interface fournie par le service substitut Vendeur 2 décrit le comportement requis par un consommateur pour placer une commande de manière différente que celle décrite par le Vendeur 1. En effet, les lignes de la commande doivent être envoyées en un seul message Commande, dont la structure regroupe toutes les lignes de la commande. La compatibilité

entre l'interface du service Consommateur et celle du service Vendeur 2 n'est pas vérifiée car le consommateur ne peut pas envoyer un seul message contenant l'ensemble des lignes de sa commande.

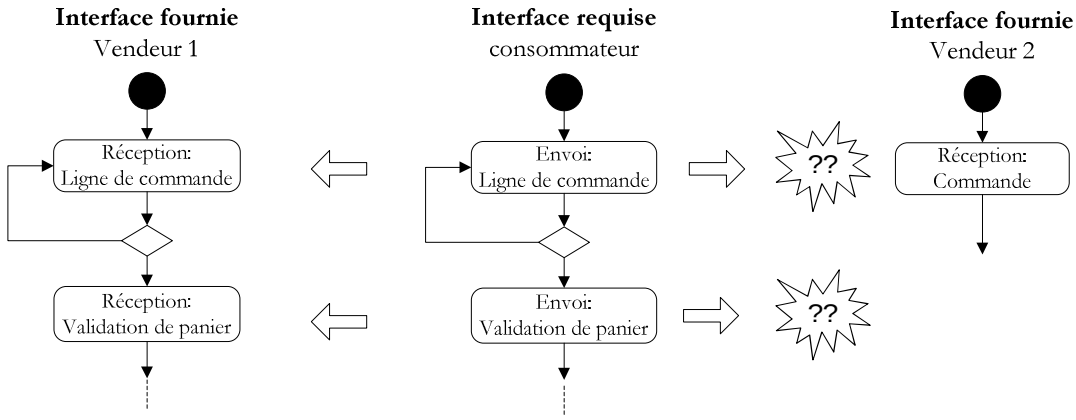

Figure 1.8 n exécutions d'une opération versus une exécution unique d'une autre

La structure des messages envoyés par le Consommateur, ainsi que le comportement de celui-ci diffèrent de la structure de messages et du comportement que le Vendeur 2 exige. Ce type d'incompatibilité est liée à la fois aux aspects structurels et comportementaux des interfaces des services. Plus précisément, la différence entre la structure de l'interface du fournisseur substitut et celle du consommateur, a engendré une différence comportementale entre celles-ci.

Une exécution unique d'une opération *versus* des multiples exécutions d'une autre opération

Dans la figure 1.9, l'interface fournie par le service Vendeur 1 décrit la réception d'un message Commande envoyé par un servie Consommateur. La structure de ce message collectionne toutes les lignes d'une commande. L'interface du consommateur est compatible à cette interface.

D'après l'interface fournie par le service substitut Vendeur 2, le consommateur doit envoyer séparément les lignes d'une commande en appelant n fois l'opération Réception : Ligne de commande. Etant donné que l'interface du service Consommateur ne correspond pas, ni d'un point de vue structurelle, ni d'un point de vue comportementale à celle du service fournisseur 2, une incompatibilité s'est manifesté dans les interactions entre les deux services. Notamment, le Consommateur ne peut pas envoyer plusieurs lignes de commande dans différents messages.

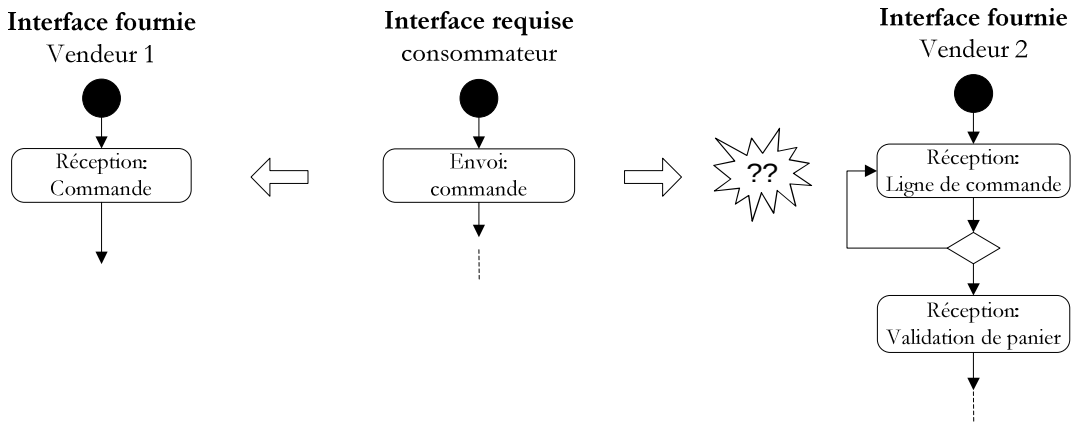

Figure 1.9 Une exécution unique d'une opération versus n exécutions d'une autre

Une exécution d'une opération *versus* une exécution de deux ou plusieurs opérations différentes les unes des autres

Dans la figure 1.10, l'interface fournie du service Vendeur 1 décrit le comportement requis par un consommateur pour envoyer l'adresse de facturation ainsi celle de livraison de sa commande. Celles-ci doivent être envoyées en un seul message. L'interface du service Consommateur est compatible à l'interface du service Vendeur 1.

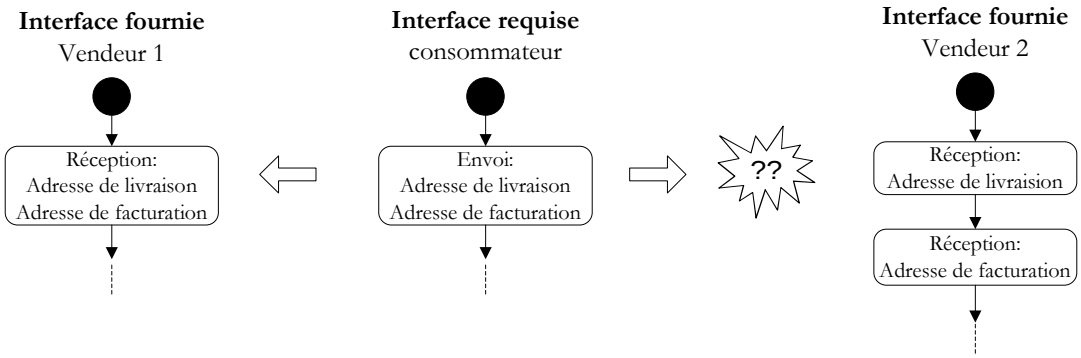

Figure 1.10 Une exécution d'une opération versus une exécution de deux ou plusieurs

D'après l'interface fournie du service Vendeur 2, le consommateur doit envoyer séparément les deux adresses en appelant l'opération Réception : Adresse de livraison, suivie de l'opération Réception : Adresse de facturation.

Etant donné que le service Consommateur n'est pas en mesure d'envoyer les deux adresses en un seul message, les interactions entre celui-ci et le Fournisseur 2 n'aboutissent pas.

Une exécution de deux ou plusieurs opérations différentes les unes des autres *versus* une exécution d'une opération

Dans la figure 1.11, l'interface fournie du service Fournisseur 1 décrit le procédé de paiement d'une commande. Le consommateur doit envoyer un message sollicitant l'opération Réception : Informations personnelles, suivie d'envoi d'un message invoquant l'opération Réception : Informations bancaires. Dans cet exemple, l'interface du service Consommateur est compatible à celle du Vendeur 1.

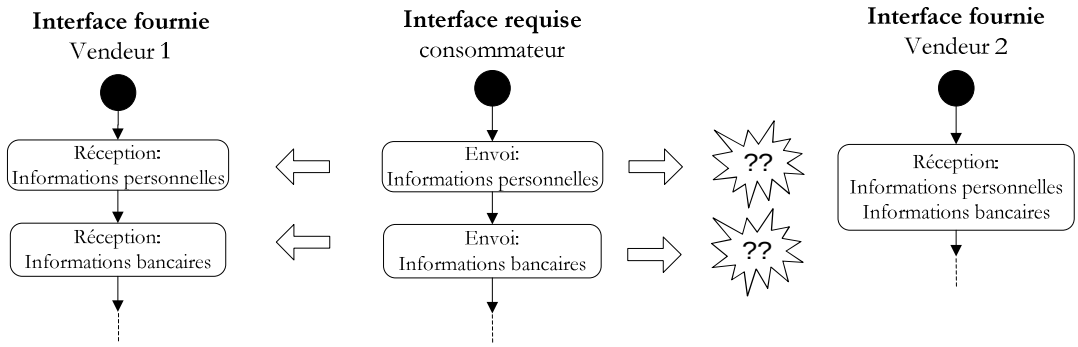

Figure 1.11 Une exécution de ou plusieurs opérations versus une exécution d'une autre

D'après son interface fournie, le service Vendeur 2 fournit une seule opération pour effectuer le paiement. Celle-ci prévoit la réception d'un seul message contenant les informations personnelles du consommateur et ses informations bancaires. Les interactions entre le service Consommateur et le service Fournisseur 2 n'aboutissent pas car le premier ne peut pas envoyer le message requis par le deuxième.

1.3.3 Objectifs de la thèse

Les recherches que nous menons dans ce travail de thèse visent à résoudre les problèmes des incompatibilités mixtes dans les interactions entre deux services consommateur et fournisseur. Pour ce faire, l'approche de résolution par la mise en place d'adaptateur a été adoptée.

Les objectifs de ma thèse sont d'une part de proposer une approche générique pour la génération et le déploiement automatique d'un adaptateur entre deux services en s'appuyant sur leurs descriptions structurelles et comportementales. D'autre part de proposer un cadre permettant une substitution transparente et flexible d'un service fournisseur par un autre vis-à-vis du consommateur du premier. C'est-à-dire, sans que le propriétaire du service consommateur se préoccupe des détails de sélection dynamique du service substitut, et de la

liaison (en anglais : *binding*) à celui-ci.

Nous avons décomposé le premier objectif en :

- Fournir des mécanismes de détection des incompatibilités mixtes entre l'interface du service consommateur et celle du service fournisseur,
- Définir une méthode de résolution générique pour chacune des incompatibilités détectées,
- Fournir des mécanismes d'adaptation des incompatibilités lors des interactions entre deux services. Il s'agit d'intercepter les messages échangés entre les services de manière à réaliser la méthode de résolution correspondante, dès lors qu'une incompatibilité est détectée.

Le second objectif est quand à lui décomposé en :

- Alléger le couplage entre un service consommateur et un service fournisseur en s'appuyant sur des mécanismes d'abstraction d'interfaces.
- Fournir des mécanismes permettant à un service consommateur d'accéder à travers son interface à un ensemble de services répondant à ses besoins fonctionnels, d'en sélectionner un, et d'interagir avec sans avoir besoin de connaître son interface réelle.

1.4 Contributions de la thèse

Dans cette thèse, notre contribution s'étend, tant sur le plan théorique que sur le plan pratique. Il s'agit d'une part d'un canevas pour la génération automatique des adaptateurs par détection et résolution automatique des incompatibilités [TABFB09, TFDB08] (voir section 1.4.1). D'autre part, nous proposons une architecture logicielle fournissant un cadre à la substitution de services [TBFM06, BMT+07, FDTB07] (voir section 1.4.2).

1.4.1 Canevas pour la génération automatique des adaptateurs

Dans cette section nous présentons brièvement notre contribution pour la génération automatique des adaptateurs. Cette contribution s'appuie sur la détection des incompatibilités mixtes entre deux interfaces de deux services, ainsi la génération et le déploiement d'un adaptateur qui permet de les résoudre à l'exécution. Dans notre approche, la détection des incompatibilités ainsi que la génération de l'adaptateur sont automatiques.

En premier lieu, une modélisation d'interface structurelle et comportementale des services

sous forme d'automates a été adoptée. Dans cette modélisation, le comportement *observable*, ainsi que les descriptions structurelles associées ont été considérés. En d'autres termes, il s'agit de modéliser la séquence des opérations d'envoi et de réception de messages, ainsi que les descriptions structurelles de ces derniers décrites dans les interfaces.

Dès lors que les interfaces de deux services consommateur et fournisseur sont décrites respectivement en deux automates, une étape de détection des incompatibilités entre ceux-ci est réalisée. L'algorithme de détection permet d'identifier les incompatibilités mixtes discutées à la section précédente. D'autres types d'incompatibilités, telles que les incompatibilités sémantiques et comportementales sont supposées résolues par ailleurs [MRI07, AB08].

Pour la résolution des incompatibilités détectées, un adaptateur d'interactions est mis en place. Celui-ci est responsable d'intercepter les messages échangés entre deux services afin de résoudre leurs incompatibilités. Notre approche pour la génération de l'adaptateur est pilotée par le modèle d'automate. Cela permet de modéliser l'adaptateur indépendamment de son implémentation cible, permettant ainsi une bonne réutilisation des modèles.

Plus précisément, nous avons spécifié pour chaque type d'incompatibilité un patron de résolution, exprimé par un modèle d'automate configurable et composable. Ainsi, la construction automatique de l'adaptateur s'appuie sur la configuration du patron correspondant en fonction de paramètres (messages à consommer, messages à produire) de l'incompatibilité détectée. L'adaptateur généré entre deux services est la composition de différents patrons associés aux incompatibilités détectées entre les interfaces de services.

Une fois généré, l'automate de l'adaptateur suffisamment détaillé est projeté vers un modèle spécifique. Les caractéristiques de fonctionnement et de comportement de l'adaptateur qui ont été spécifiées de façon générique, sont alors transformées pour tenir compte des spécificités de la plate-forme cible, qui est dans notre cas une plate-forme mettant en œuvre la technologie de traitement des événements complexes[10] *(Complex Event Processing, CEP)*. Cette transformation s'effectue à l'aide de composants cartouches (en anglais : *templates*) que nous avons mis en œuvre. Chaque cartouche étant conçue pour générer du code exécutable en termes des requêtes continues pour la technologie *CEP* (voir chapitre 5).

La figure 1.14 présente l'architecture du canevas *AdaptTer*, mettant en œuvre notre contribution pour la génération automatique des adaptateurs.

[10] http://complexevents.com/

Le canevas comporte deux environnements différents : un environnement de conception (partie basse de la figure), et un environnement de déploiement (partie haute de la figure).

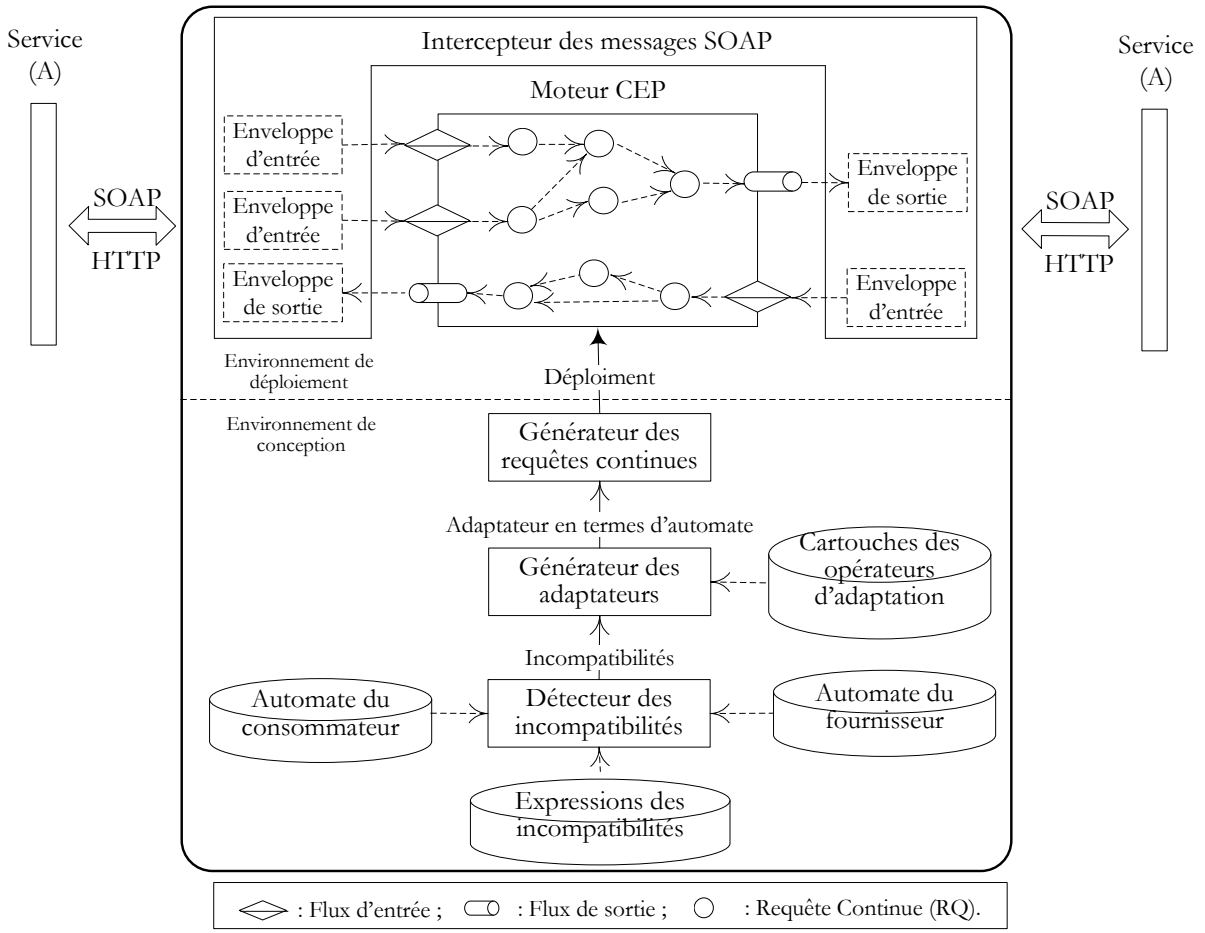

Figure 1.12 Architecture du canevas

L'environnement de conception comprend les modules ci-après :

- Le Détecteur des incompatibilités. permet de comparer deux automates décrivant respectivement un service consommateur et un autre fournisseur afin de détecter leurs incompatibilités.

- Le Générateur des adaptateurs. s'appuie sur le résultat de la détection des incompatibilités pour générer automatiquement l'automate qui spécifie le fonctionnement et le comportement de l'adaptateur.

- Le Générateur des requêtes continues. S'appuie sur le modèle de l'adaptateur exprimé en automates. Il traduit celui-ci en des requêtes continues à l'aide des cartouches d'opérateurs d'adaptation. Par exemple, dans le cas d'une incompatibilité de type

Multiples exécutions d'une opération versus une exécution unique d'une autre opération, illustrée sur la figure 1.8, la requête continue générée est exécutée en continu en attendant les messages provenant du consommateur. Son principe consiste à consommer et stocker les lignes de commande envoyées un par un par le consommateur jusqu'à la réception du message Validation de panier qui annonce la terminaison de la commande. A ce stade, les lignes stockées sont corrélées et envoyées en un seul message au service fournisseur.

L'environnement de déploiement comprend les modules ci-après :

- Le Moteur CEP. héberge et exécute les requêtes continues générées. Il fournit le service de réception, de corrélation, d'analyse et de traitement des messages en fonction des requêtes déployées.

- L'Intercepteur des messages SOAP. est chargé de capter les messages provenant du service fournisseur, puis de les acheminer à travers le moteur CEP afin d'être adaptés, enfin de transporter au fournisseur les messages ainsi obtenus.

1.4.2 Architecture logicielle facilitant la substitution de services

Dans cette section, nous présentons brièvement notre contribution pour le développement des applications à base de services permettant une substitution flexible entre ceux-ci.

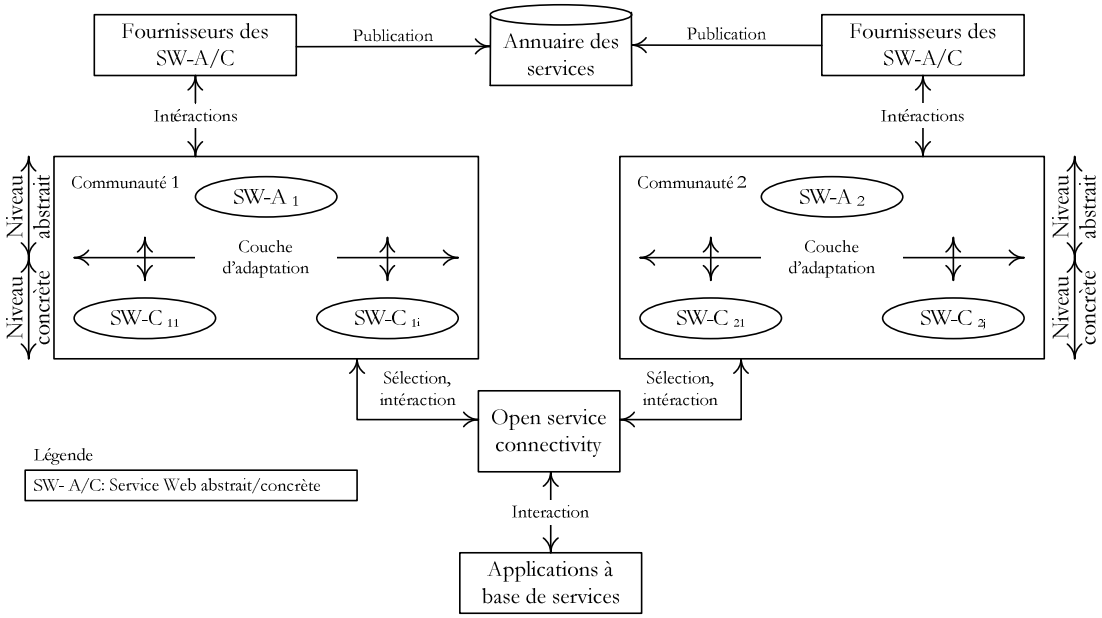

Figure 1.13 Architecture de l'approche

La notion de communauté de services a été introduite pour formaliser le rapport entre les

niveaux abstrait et concret de notre approche [TFDB08]. Une communauté de services est perçue comme un moyen d'exposer des descriptions communes d'un caractère fonctionnel désiré sans explicitement se référer à un service spécifique. Dans la figure 1.13, un Fournisseur de communauté initialise le regroupement d'un ensemble de services similaires, qualifiés de concrets, et les expose par le biais d'une interface qualifiée d'abstraite. Celle-ci offre une vue unifiée de l'ensemble des services regroupés dans la communauté. Un consommateur d'une communauté est conçu sur la base de l'interface abstraite de celle-ci. A l'exécution, l'interface abstraite n'implémente pas sa fonctionnalité, mais les requêtes de ses consommateurs sont déléguées à un ou plusieurs de ses services concrets. Grâce à la Couche d'adaptation entre l'interface abstraite et les interfaces de services concrets, le consommateur peut interagir avec tout service de la communauté sans avoir besoin de connaître son interface. Ainsi, la substitution d'un service concret par un autre est transparente pour le consommateur.

L'OSC est un composant logiciel dont l'objectif est d'appliquer le principe de pilotes ODBC et JDBC (utilisés dans des applications de bases de données) dans un environnement à base de services. Plus précisément, il est responsable de gérer les interactions entre les consommateurs des communautés d'une part et les communautés d'autre part. Il fournit une API des fonctions permettant aux consommateurs la sélection d'un service, son exécution, ainsi que sa substitution par un autre d'une façon flexible et transparente, c'est-à-dire sans que le consommateur se soucie des détails de la substitution (sélection de service, résolution des incompatibilités, etc.).

1.5 Plan de la thèse

Ce document est structuré de la manière suivante :

Dans la première partie de la thèse nous présentons en deux chapitres le contexte de recherche et un état de l'art. Dans le chapitre 1, après avoir présenté le contexte de recherche de la thèse, nous identifions la problématique liée aux incompatibilités dans les interactions entre un service consommateur et un service fournisseur. Les incompatibilités traitées proviennent de la différence dans les définitions d'interfaces structurelles et comportementales de services. Dans le chapitre 2, nous présentons un état de l'art des travaux qui s'apparentent à notre problématique. En particulier, nous présentons les différentes approches sur le test de compatibilité des interfaces et la résolution des incompatibilités identifiées lors de ces tests. Nous avons classifié les approches présentées selon la dimension des incompatibilités traités, à savoir : structurelle, comportementale, et mixte.

Dans la deuxième partie de la thèse, nous présentons notre proposition pour l'adaptation des interactions de services par génération automatique des adaptateurs. Le chapitre 3 introduit une modélisation des interfaces structurelles comportementales des services, en se basant sur les automates déterministes en nombre fini d'états. Le chapitre 4 présente la génération automatique des adaptateurs qui constitue la contribution principale de cette thèse. En particulier, nous détaillons le principe de notre solution pilotée par les modèles (patrons) pour la détection et la résolution des incompatibilités. Le chapitre 5 présente la mise en œuvre des adaptateurs générés dans une infrastructure de traitement des événements complexes *CEP*. Puis, nous présentons les détails d'implantation du canevas proposé, *Adapter*.

Dans la troisième partie, composée du chapitre 6, nous présentons notre approche à base de communautés de services, et du composant *OSC*, pour faciliter la substitution entre des services similaires.

Dans la dernière partie, le chapitre 7 termine cette thèse par une conclusion sur ses contributions, mais aussi pour ses limites, et discutent des perspectives de recherche envisagées.

CHAPITRE 2

Etat de l’art

Sommaire

2.1 Introduction

Dans le chapitre introductif de ce manuscrit, nous avons précisé le contexte de notre travail de recherche et nous avons identifié la problématique à laquelle nous nous intéressons. Il s’agit de la problématique de la détection et de la résolution des incompatibilités entre les interfaces des services. Dans ce chapitre, nous proposons un état de l’art sur les différentes méthodes et approches proposées autour de cette problématique.

Dans la section 2.2, nous discutons des différents types des incompatibilités qui peuvent avoir lieu entre des interfaces de services, à savoir des incompatibilités structurelles, comportementales, et mixtes. Dans la section 2.3, nous présentons des approches proposées pour les tests de compatibilité structurelle et comportementale des interfaces de services. Dans la section 2.4, nous présentons des approches portant sur la résolution des incompatibilités détectées entre deux interfaces. Dans les sections 2.3.2 et 2.4.4 nous proposons une synthèse sur les différentes approches proposées pour les tests de

compatibilités et la résolution des incompatibilités. Cette synthèse permet de mettre en évidence non seulement les apports de ces approches mais aussi leurs limites par rapport à des critères qui renvoient aux finalités de cette thèse.

2.2 Incompatibilités des interfaces des services

Plusieurs approches ont été proposées pour résoudre les incompatibilités dans les interactions de services Web. L'étude et la classification des incompatibilités fait l'objet de plusieurs travaux [NBCT06, ATP06, MRI07]. Ces travaux distinguent entre les incompatibilités structurelles (voir la section 2.2.1) et comportementales (voir la section 2.2.2). D'autres travaux tels que [BCG+05, DSW06] traitent des incompatibilités mixtes (voir la section 2.2.3).

2.2.1 Incompatibilités structurelles

Les incompatibilités structurelles se produisent dans les interactions entre deux services lorsque la structure du message envoyé par le premier est différente de celle du message reçu par le deuxième.

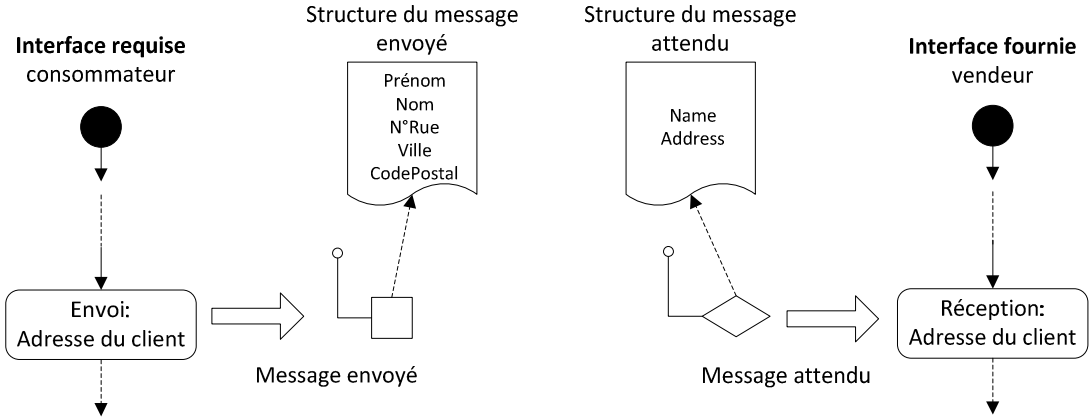

Figure 2.1 Incompatibilité structurelle

La figure 2.1 illustre un exemple d'incompatibilité structurelle. La structure du message envoyé par le service Consommateur comprend cinq attributs à savoir, Prénom, Nom, Numéro de rue, Ville, et Code postal, alors que la structure du message attendu par le service Vendeur regroupe le nom et le prénom du client dans un attribut nommé Name, ainsi que le numéro de rue, la ville, et le code postal dans un autre attribut nommé Address. Cette différence entre les structures de message, se traduit une incompatibilité dans les interactions entre les deux services : Le message envoyé par le service Consommateur est rejeté par le

service Vendeur car sa structure ne correspond pas à celle du message attendu.

2.2.2 Incompatibilités comportementales

Les incompatibilités comportementales se produisent dans les interactions entre deux services lorsque la séquence d'exécution d'opérations du premier service n'est pas cohérente avec celle des opérations du deuxième service.

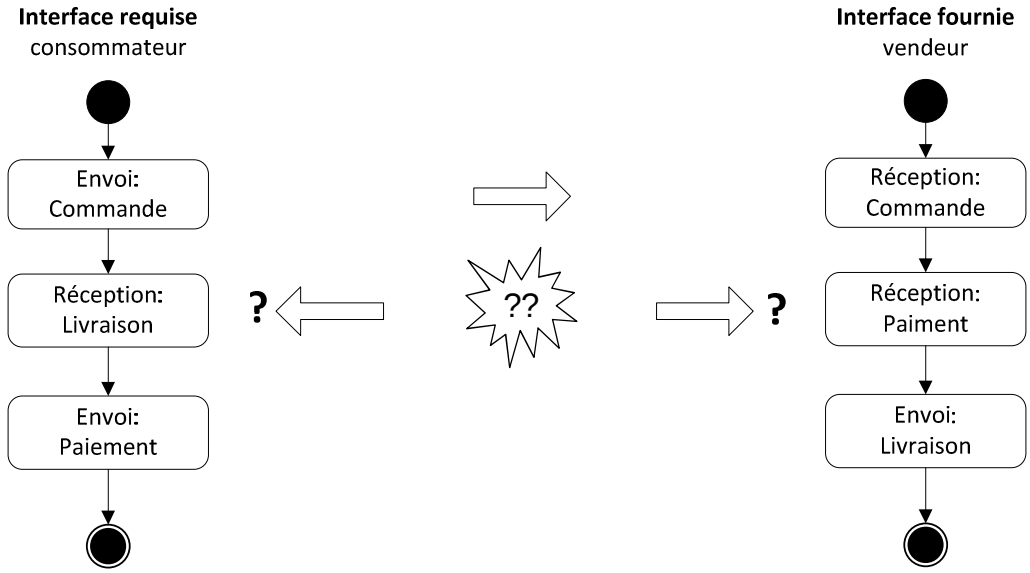

Figure 2.2 Incompatibilité comportementale

La figure 2.2 illustre un exemple d'incompatibilité comportementale. Après l'envoi d'une commande (Envoi : Commande), le consommateur prévoit la réception d'un message confirmant la livraison (Réception : Livraison) avant d'effectuer le paiement (Envoi : Paiement). Cependant, après la réception de la commande du consommateur, le fournisseur s'attend à la réception du message du paiement (Réception : Paiement) avant d'envoyer le message de Livraison (Envoi : Livraison). Donc, après la mise en place de la commande, l'interaction entre le fournisseur et le consommateur est en état d'interblocage car le vendeur s'attend à la réception du message de paiement de la part du consommateur, et ce dernier s'attend à la réception du message de livraison de la part du vendeur.

2.2.3 Incompatibilités mixtes

Dans certaines situations, une différence dans la structure des messages se traduit par une différence dans la séquence des envois et des réceptions des nouveaux messages, et inversement. Par exemple, lorsqu'un message envoyé par le consommateur contient plusieurs

occurrences de fragments de messages de même structure (une commande contient plusieurs lignes de Commande), alors que le message attendu par le fournisseur ne peut contenir qu'un seul fragment. Cette incompatibilité entre les structures de messages provoque une incompatibilité comportementale. Pour que les n fragments du message envoyé soient reçus par le consommateur, il est nécessaire de fractionner le message en n sous-messages de manière que chacun contient un fragment du message original, puis envoyer les fragments un par un au service fournisseur. La liste complétée des ces incompatibilités est donné dans la section 1.3.2 du chapitre 1.

2.3 Tests de compatibilité des interfaces

Plusieurs approches ont été proposées pour étudier la compatibilité des interfaces des services. Certaines approches se concentrent sur les interfaces structurelles de services, d'autres approches se focalisent sur les interfaces comportementales. Dans la section 2.3.1, nous présentons des approches portant sur le test de compatibilité entre les interfaces structurelles. Dans la section 2.3.2, nous discutons des approches se focalisant sur le test de compatibilité entre les interfaces comportementales. La section 2.3.3 fournit une synthèse d'approches présentées.

2.3.1 Test de compatibilité structurelle

Le test de compatibilité structurelle entre deux services consiste à vérifier la cohérence sur le plan structurel des messages échangés par les opérations de ces services. Ce test revient à comparer les descriptions *WSDL* dans lequel les messages envoyés et reçus sont spécifiés.

Étant donné que l'interface *WSDL* de services Web est basée sur le standard *XML*, les approches pour tester la compatibilité structurelle s'appuient sur les techniques de comparaison de schémas *XML*. Certaines de ces approches utilisent des techniques classiques [ZYGW06, WS03, DHM+04, WW05, KBH+04], d'autres approches s'appuient sur des techniques étendues avec l'utilisation des ontologies [NM03, MVR+04, HMO05].

Techniques classique de comparaison de schémas XML

Zhang et coll. [ZYGW06], considèrent un service comme étant un ensemble d'opérations dont chacune est implémentée d'une fonction déterminée. Une opération est spécifiée par un nom, et les types de données de ses messages d'entrée/sortie.

Afin de vérifier la compatibilité structurelle de deux services, les auteurs proposent un algorithme qui consiste à comparer les opérations de ceux-ci telles qu'elles sont décrites dans

les interfaces *WSDL*. La comparaison de deux opérations induit à la comparaison de leurs messages d'entrée/sortie. Ce qui se traduit par la comparaison de types d'objets communiqués par ces messages.

Les types de données de ces objets, décrits dans un document *WSDL*, sont des éléments *XML*. Dans un schéma *XML*, trois types de données peuvent être distingués, à savoir : primitif, simple et complexe. L'algorithme proposé permet de comparer non seulement les types primitifs et simples de données mais aussi les types complexes.

Au sujet de la comparaison des types de valeurs de l'élément *(valeur des attributs type)*, l'algorithme considère deux types de valeurs comme compatibles au cas où ils sont identiques ; semi-compatible à condition qu'ils puissent être adaptés l'un par rapport à l'autre, c'est le cas de int et float ; incompatible dans le cas où aucune adaptation n'est possible, c'est le cas de int et string.

En ce qui concerne les types complexes, l'algorithme propose de comparer l'ensemble des sous-éléments du premier, avec l'ensemble des sous-éléments du second. Les paires d'éléments ayant le score de similarité maximal sont renvoyées.

L'originalité de l'approche est qu'elle ne nécessite aucune annotation sémantique pour appuyer le processus de comparaison. L'algorithme proposé a montré une bonne efficacité pour déterminer la similarité entre les types de données des messages échangés. Cependant, l'approche n'est pas en mesure de déterminer la similarité sémantique entre deux objets. Notamment, l'algorithme ne détecte pas la similarité entre deux messages de deux opérations libellés avec deux nom sémantiquement compatibles, tels que Placer une Commande et Place an Order.

Dans [WS03], Wang et coll. proposent un algorithme pour tester la compatibilité d'un service sélectionné avec un autre dans le cadre d'une composition de services. De même que l'approche présentée ci-dessus, les auteurs décomposent un service, décrit dans l'interface *WSDL*, en termes d'opérations et de messages d'entrée/sortie.

Ainsi, pour étudier la compatibilité de deux services, ils proposent une méthode constituée de quatre étapes principales :

Comparaison de type des données : cette étape consiste à comparer les types de données décrits dans les documents *WSDL* des deux services. Les résultats de cette comparaison sont retournés sous forme d'une matrice qui énonce les degrés de similarité entre les différents types du premier service et ceux du second.

Calcul de similarité entre des messages : cette étape consiste à calculer et retourner sous forme d'une matrice les degrés de similarité entre les différents messages d'entrée/sortie décrits dans les documents *WSDL* de services concernés. En effet, le degré de similarité entre deux messages est calculé d'après le degré de similarité entre leurs types de données, obtenu en première étape.

Calcul de similarité entre des opérations : cette étape consiste à déterminer et retourner sous forme d'une matrice les degrés de similarité entre les différentes opérations décrites dans les documents *WSDL*. Le degré de similarité entre deux opérations est calculé d'après le degré de similarité de leurs messages d'entrée/sortie, obtenu à l'étape précédente.

Calcul de similarité entre les deux services : cette étape consiste à déterminer le degré de similarité globale entre les deux services. Celui-ci est obtenu en identifiant d'abord les paires d'opérations dont le degré de similarité est maximal dans chaque ligne de la matrice. Sur la base de cette moyenne, les deux services sont considérés compatibles ou non.

Ce travail s'est montré efficace à découvrir les correspondances entre deux opérations dont les messages envoyés et reçus sont de même type de données. Cependant, des opérations ayant des messages de même type de données avec des noms de message différents sont hors du champ de l'algorithme proposé.

Dans [DHM+04], Dong et coll. proposent un moteur de découverte de services Web basé sur une technique de comparaison d'interfaces. Le moteur prend en entrée une interface soumise par l'utilisateur, et retourne une séquence d'interfaces triée par ordre croissant selon les degrés de similarité avec l'interface soumise. Alors que les approches présentées ci-dessus [ZYGW06, WS03] reposent sur le type de données de messages dans leur test de similarité, le présent travail repose sur le nom de messages. L'algorithme proposé consiste en deux étapes. La première étape consiste à regrouper les noms de messages en des concepts sémantiquement significatifs. La deuxième étape consiste à comparer les types de données de messages appartenant au même concept.

Le regroupement des messages selon leur nom n'utilise pas d'ontologie, mais cela s'appuie sur l'algorithme NGram [NG09], qui permet de calculer la similarité entre deux chaines de caractères en fonction de *q-grams* (chaine de caractères) partagés entre les deux chaines. L'originalité de cette approche est qu'elle touche aux aspects sémantiques de messages sans utiliser d'ontologie.

Dans [WW05], Wu et coll. proposent de classifier les propriétés de services Web en quatre catégories, à savoir : 1) les propriétés communes tel que le nom du service, la description du service, la clé du service, etc. 2) les propriétés spéciales qui comportent des

propriétés spécifiques à un certain type de services, par exemple les services d'émission possède une propriété qui définit le type de média qu'il fournit, 3) l'interface du service, et 4) la qualité de service. Pour chaque catégorie, une méthode de calcul de similarité a été donnée. Ainsi, la méthode de calcul de similarité d'interfaces est basée sur la comparaison de messages d'entrée/sortie d'opérations. La comparaison de messages est à son tour basée sur la comparaison du nom et du type de données de messages. La comparaison de nom de messages est basée sur l'algorithme NGram [NG09].

Kawamura et Col. [KBH+04] présente un autre système, dans lequel une série de trois filtres sont utilisés pour affiner la comparaison entre les interfaces de services. Ces filtres couvrent les noms d'opérations et de messages de services, les types de données associées aux messages d'entrée/sortie, et la relation de subsomption des contraintes sur ces messages.

Techniques de comparaison étendues avec l'utilisation des ontologies

Les ontologies et le domaine du web sémantique se situent à un niveau différent de ce que nous venons de voir. En effet, les ontologies font partie des systèmes de raisonnements permettant, par exemple, de faire des déductions automatiques à partir d'une description formelle.

Dans le cadre du web sémantique, il est courant d'utiliser des services décrits avec *DAML-S*. *DAML-S* [MBD03] un langage reposant sur *XML* et *RDF(S),* très utilisé dans le domaine de l'intelligence artificielle où tout doit être décrit le plus formellement possible pour permettre la réutilisation automatique des éléments ainsi décrits.

Les auteurs de [NM03] utilisent ce langage dans le but de décrire, simuler, composer, tester et vérifier les compositions de services web. Pour cela, ils ont défini une sémantique pour une partie des éléments de *DAML-S* (constructeurs de séquence, de concurrence, etc.) en termes de logique du premier ordre *(situation calculus).*

Afin de prendre en compte les aspects sémantiques liés aux noms des opérations et des messages lors du calcul de similarité entre deux interfaces structurelles, Miller et Coll. [MVR+04] proposent d'annoter l'interface *WSDL* avec des extensions relatives aux opérations et messages d'entrée/sorties des services. Ces extensions contiennent des références aux concepts décrits dans les ontologies de description du domaine de connaissance associé au service Web, afin de spécifier la sémantique de messages, mais aussi les pré-conditions et les effets des opérations. *WSDL-S* vise à fournir une approche *allégée* d'annotation sémantique de services Web.

Dans [HMO05] Haller et coll. proposent d'annoter les descriptions WSDL de services

Web à l'aide des concepts définis dans une ontologie qui décrit le domaine des services. En supposant que toutes les informations nécessaires sont décrites dans l'ontologie commune, un algorithme qui permet d'identifier les similarités entre les concepts des ontologies a été mis en œuvre. En plus de la détection de similarités entre deux interfaces, cet algorithme permet de générer des correspondances entre ces dernières. Un composant spécifique effectue la transformation de données durant les interactions entre les services.

2.3.2 Test de compatibilité comportementale

Dans les approches que nous allons présenter, le principe de solution consiste à modéliser l'interface comportementale de services Web, fournie en BPEL, BPMN, WSCI, etc. à l'aide des outils formels, tels que : des automates, l'algèbre de processus, etc.

Techniques à base d'automates

Par définition, un automate déterministe en nombre fini d'états est une machine abstraite dont le principe est de composer le comportement du système en nombre fini d'états. A partir de chaque état il est alors possible d'effectuer une transition vers un autre état. Il est donc nécessaire de déterminer les conditions de transitions (définissant ainsi les arcs dans un graphe d'états). Ces conditions de transition d'états peuvent dépendre de paramètres déterminants ou aléatoires, cependant ils doivent tous être accessibles depuis l'état en cours.

Un automate permet de modéliser le comportement *observable* d'un service Web. Dans cette modélisation, les états représentent les différentes phases dans lesquelles un service peut passer pendant son interaction avec un client. Chaque interaction est considérée comme une transition. Les transitions sont étiquetées par les actions correspondantes. Une action d'envoi de message est représentée par le symbole *! (!(msg) représente une action d'envoi du message msg)*. Une action de réception de message est représentée par le symbole *? (?(msg) représente une action d'envoi du message msg).*

La figure 2.3 illustre un exemple où un processus de gestion de commande est représenté sous forme d'un automate. Les rectangles représentent les états de l'automate (Vérification d'utilisateur, Réception de Commande, Validation de commande). Les flèches représentent les transitions qui partent d'un état source vers un état cible et qui sont étiquetées d'une action d'envoi ou de réception d'un message. A titre d'exemple, la flèche qui relie l'état source Réception de commande, à l'état cible Validation de commande est étiquetée par l'action de réception de message ?(Valider Panier) qui signale la validation de la commande de la part de l'utilisateur. Une garde peut être associée à une transition. Une garde représente une condition qui doit être vérifiée pour déclencher la transition. L'état initial est représenté par une boule de

couleur blanche avec un contour noir, et l'état final est représenté par une boule de couleur noire.

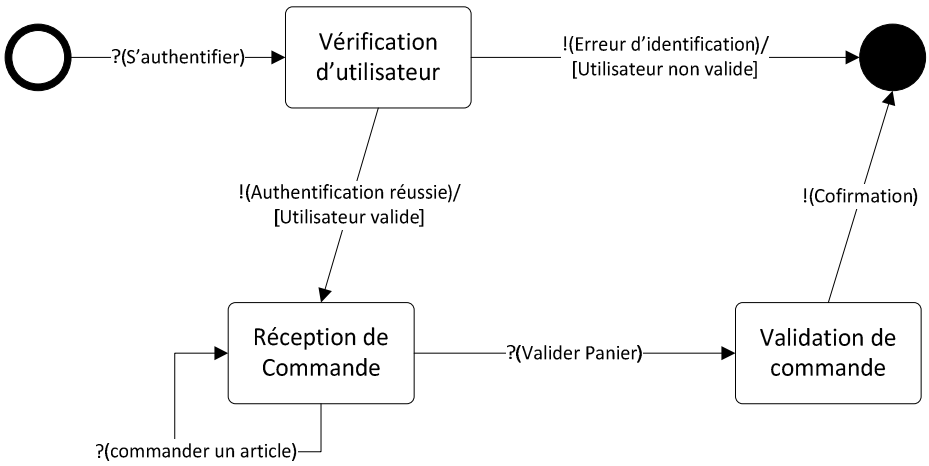

Figure 2.3 Représentation en Automate d'un processus de commande

Dans cet exemple, après la réception d'un message d'identification d'un client, l'automate passe de l'état initial à l'état de Vérification d'utilisateur. Si l'identification du client échoue, un message d'erreur sera renvoyé à l'utilisateur !(Erreur d'identification) et le processus de commande est terminé. Dans le cas contraire, un message qui signale la bonne identification lui sera envoyé, et l'automate passe dans l'état Réception de commande. Dans cet état, l'automate attend soit la réception d'un message de commande d'articles, soit la réception d'un message de validation de panier. A la réception d'une commande d'articles, l'automate reste toujours dans l'état Réception de Commande car l'état cible de la transition étiquetée par l'action ?(Réception de commande) est l'état source lui-même. D'ailleurs, à la réception de message de validation de panier ?(Valider Panier), l'automate passe à l'état Validation de commande. Puis, un message de confirmation de la bonne réception de la commande est envoyé au client !(Confirmation) et l'automate passe à l'état final ce qui termine le processus.

L'un des principaux apports des approches à base d'automates réside dans la proposition d'algorithmes de vérification de compatibilité d'interfaces de services. Ces algorithmes basés sur des fondements mathématiques, permettent de supporter une vérification rigoureuse de la compatibilité d'interfaces de services.

Wombacher et coll. [WFMN04] proposent de modéliser l'interface comportementale des services à l'aider d'automates déterministes en nombre fini d'états, puis de s'appuyer sur l'opérateur d'intersection d'automates afin de vérifier la compatibilité entre les interfaces comportementales de services. En effet, deux automates sont considérés compatibles si et seulement si l'ensemble d'enchainements d'envois et de réceptions de messages, obtenu par

leur intersection est accepté par les deux automates.

Dans la figure 2.4, les automates Vendeur et Consommateur décrivent respectivement les interfaces comportementales de deux services vendeur et consommateur. L'automate I représente leur intersection. Les nœuds blancs représentent les états d'un service ; les nœuds noirs représentent les états finaux. Les transitions entre états sont libellées avec les notations suivantes : source#destinataire#nom_message, où source représente l'émetteur de message, destinataire représente le récepteur de message, et nom_message est le nom de message.

Selon l'interface du service vendeur, Le processus de vente démarre à la réception d'un ordre d'achat envoyé par un consommateur (C#V#PO). Ensuite un message de livraison est envoyé au consommateur (V#C#Livraison). Puis, deux choix de paiement sont proposés au consommateur, soit par carte de crédit (C#V#CC), soit par chèque (C#V#Chèque). Au cas où le produit commandé n'est pas disponible dans le stock, le vendeur rejette la commande par l'envoi du message (V#C#non-Disponible).

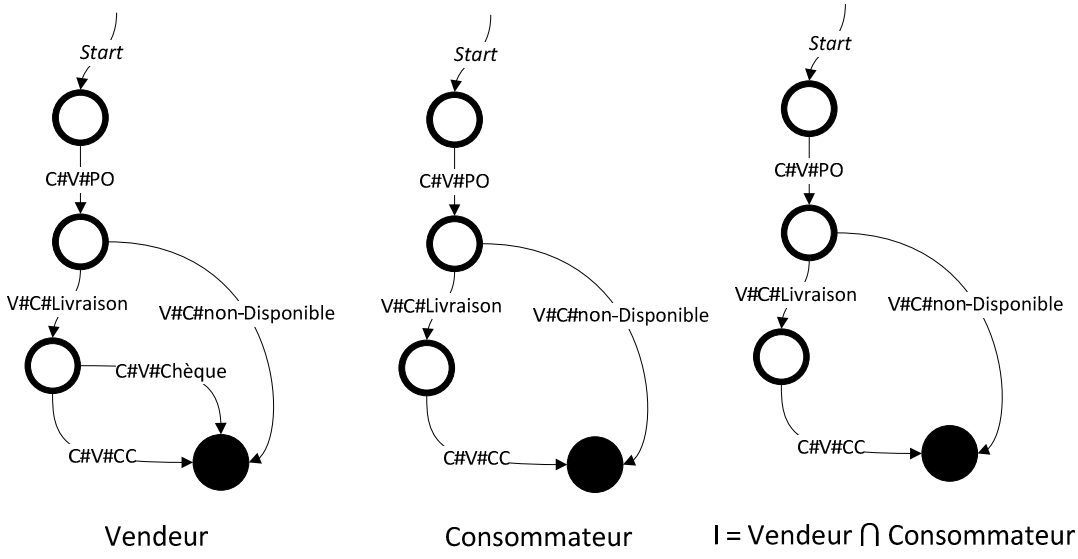

Figure 2.4 Automates décrivant l'interface comportementale des services

L'automate I, intersection de Vendeur et Consommateur, comprend des enchaînements de transitions acceptés par les deux automates, donc ceux-ci sont considérés compatibles.

Dans [BSBM04], Bordeaux et coll. propose d'étudier la compatibilité entre deux interfaces en s'appuyant sur la notion d'automates. Les auteurs proposent une solution dont le principe consiste à modéliser l'interface comportementale des services à l'aide d'automates, puis à introduire des méthodes pour vérifier la compatibilité entre les automates. Selon les auteurs, un service donné A est dit compatible à un autre service B sur le plan comportemental

si et seulement si à chaque action d'envoi de message *(respectivement de réception de message)* décrite dans l'interface I_A correspond une action de réception de message *(respectivement d'envoi de message)* décrite dans l'interface I_B, et que l'enchainement d'exécution des actions d'envoi *(respectivement de réception)* décrit dans I_A correspond à la séquence d'exécution des actions de réception *(respectivement d'envoi)* décrit dans I_B. Ainsi, une conversation entre deux services est dite cohérente si et seulement si les messages échangés entre les deux services respectent le modèle des interactions décrites dans les interfaces de ces services. Autrement la conversation est dite incohérente.

Après avoir défini la notion de compatibilité les auteurs distinguent deux types de compatibilité selon que celle-ci est dépendante du contexte, ou indépendante. La compatibilité dépendante du contexte est définie pour deux services A et A′ dans une composition donnée, si et seulement si pour tout service B compatible avec A, B est compatible avec A′. Sur la base de cette définition, A et A′ sont considérés compatible dans le contexte de la composition avec B. La compatibilité indépendante de contexte est vérifiée pour deux services A et A′ si et seulement si quelque soit un service B compatible à A, B est alors compatible à A′.

Dans [CLB08], Chae et coll. étendent la modélisation en automates des interfaces comportementales des services avec des règles de pré-condition et post-condition associées aux transitions d'automates. Une pré-condition associée à une transition (par exemple, [totale==0] !(terminer)) devrait être évaluée à vrai pour que l'action associée à la transition puisse se déclencher, et une post-condition (par exemple, ?(Paiement_total()) [total==0) est évaluée à vrai une fois que l'exécution de l'action est terminée. En se basant sur cette modélisation, les auteurs définissent la notion de trace comme étant un comportement particulier d'un service. Notamment, une trace est une séquence des transitions qui partent de l'état initial de l'automate et atteignent à l'un d'états finaux de celui-ci. Un service A est compatible avec un service A′ si et seulement si l'ensemble de traces de A comprend l'ensemble de traces de A′.

Pour vérifier de manière formelle la compatibilité comportementale entre deux services, les auteurs définissent d'abord deux notions principales, à savoir la compatibilité entre transitions et la compatibilité entre états. Une transition t_1 d'un automate A_1 est dite compatible à une autre transition t_2 d'un automate A_2, si et seulement si l'état source de t_1 est compatible à l'état source de t_2, et les pré- et post-conditions de t_1 sont cohérentes avec celles de t_2. Un état e_1 est dit compatible à un état e_2 si et seulement si pour chaque transition entrante t_1 dans e_1 il existe une transition entrante t_2 dans e_2, tel que t_1 est compatible à t_2. Les états initiaux de deux automates sont considérés compatibles car ils n'ont pas de transitions entrantes. Finalement, les auteurs définissent un automate A_2 comme compatible avec un

automate A_1 si et seulement si pour chaque transition de A_1 il existe une transition compatible de A_2.

Dans [MPC01], Mecella et coll. étudient la compatibilité d'interfaces comportementales de services dans le contexte de processus coopératifs. Par définition, un processus coopératif est spécifié comme un ensemble de services qui interagissent les uns avec les autres. Dans un processus collaboratif, il est possible de substituer un service par un autre à condition que le nouveau service respecte et garantisse la cohérence des interactions dans le processus. L'algorithme proposé pour vérifier si un service peut être remplacé par un autre dans le contexte d'un processus collaboratif. Il s'appuie sur la notion de compatibilité entre les interfaces de nouveau service et celles de services du processus. Le test de compatibilité entre les interfaces comportementales de services s'appuie sur la notion de trace. Dans ce travail, une trace est définie comme étant une séquence des actions d'envois et de réceptions de messages, acceptés par l'automate qui décrit le comportement du service concerné. Finalement, un service S_2 est considéré compatible avec S_1 si et seulement si toute trace acceptée par S_1 est aussi acceptée par S_2.

Dans [FUMK04], Foster et coll. présentent une approche pour tester la compatibilité des services au sein d'une composition de services. Pour ce faire, les auteurs proposent de transformer les descriptions d'une composition, fournie en *BPEL* en automates. Ensuite ils proposent un algorithme, qui vérifie les points suivants lors du test de compatibilité:

- La cohérence structurelle des messages échangés entre un service et son consommateur.
- L'absence d'interblocage dans l'interaction entre les services
- L'exécution de toutes les activités du processus de la première jusqu'à la dernière

Deux services, client et consommateur, qui vérifient ces conditions, sont considérés comme compatibles.

Jung et coll. [JHKK04] proposent une méthode pour l'organisation de chorégraphie de services, focalisant sur leurs contrats et modèles exécutables. Les auteurs considèrent une chorégraphie de services comme un schéma qui décrit à l'aide d'automates déterministes les séquences d'interactions entre les services dans le cadre d'un processus collaboratif. L'algorithme proposé permet d'analyser et de comparer les spécifications de deux services afin de déterminer s'il est possible ou non, de les intégrer dans une chorégraphie.

Techniques à base de l'algèbre de processus

L'algèbre de processus CCS (*Calculus of Communicating Systems process*) est une

famille de langages formels permettant de modéliser les systèmes informatiques concurrents ou distribués [MIL80]. Un processus CCS est défini comme suit: P ::= 0 | α • Q | P + Q | P || Q | P \ sm, et α ::= a?(x) | a!(x) | ρ.

Dans [BCPV04], Borgi et coll. proposent d'utiliser ce technique pour modéliser les descriptions comportementales des services Web. Dans la modélisation proposée, α représente une action de réception *(respectivement d'envoi)* de message x à travers l'opération a représentée par a?(x) *(respectivement a!(x))*. *0* représente la fin d'un processus de service. L'operateur • représente une séquence entre deux actions (d'envois ou de réceptions de messages) (α • β). L'operateur + représente un choix exclusive entre deux actions (α + β). L'operateur || représente une composition parallèle de deux actions (α || β). L'operateur \ représente une restriction entre un processus de service et un ensemble d'actions. Par exemple, P \ sm signifie que le service P peut envoyer ou recevoir des messages à travers une opération m ∈ sm à condition qu'il existe un sous processus Q qui reçoit et envoie des messages à travers m. Enfin *ρ* est utilisé pour modéliser les actions internes d'un processus.

Liu et coll. [LSZ+06] proposent un algorithme basé sur la notion de composition parallèle de l'algèbre de processus pour étudier la compatibilité entre des services. Le principe de la solution proposée consiste d'abord à modéliser la composition *C* à l'aide d'un processus CCS (C= S_1 ||…|| S_i ||… || S_n) et de s'appuyer sur la vérification de la validité de C après la substitution de S_i par S_i'. Dans le cas où la composition C′= S_1 ||…|| S_i' ||… || S_n reste valide, la substitution de S_i' par S_i est considérée comme possible. Selon les auteurs, une composition est valide si aucun message échangé entre les services de la composition ne risque d'être rejeté. La composition est valide veut dire que le nouveau service est compatible avec les autres services de la composition.

Techniques à base d'expérimentations

Contrairement au principe de modélisation proposé par les approches formelles, les techniques présentées ici consiste à appliquer des expérimentations réalisées sous forme de tests d'exécution, et d'analyse postérieure des résultats.

Pour vérifier si deux services sont compatibles, Ernest et coll. [ELP06] proposent une approche qui consiste à analyser leurs traces d'exécution postérieure. Etant donné qu'un service comporte une ou plusieurs opérations, et que chaque opération prend en entrée (respectivement, retourne en sortie) zéro ou plusieurs paramètres, l'algorithme proposé consiste à examiner chaque opération indépendamment de l'autre. Il s'agit de composer les deux services de façon que les messages de sortie du premier correspondent aux entrées du

second. L'algorithme applique plusieurs exécutions d'opérations, et pour chaque exécution si les résultats obtenus correspond à ceux attendus. Si c'est le cas, les deux services sont considérés compatibles.

2.3.2 Synthèse

Les approches existantes pour la vérification de la compatibilité entre les interfaces des services Web se concentrent soit, sur les interfaces structurelles des services soit, sur leurs interfaces comportementales. Les approches portant sur la compatibilité comportementale s'intéressent à vérifier la cohérence de la structure des messages d'envoi et de réception des opérations des services. Ces approches négligent les aspects comportementaux des opérations car elles considèrent que l'exécution d'une opération de dépend pas de celle d'une autre opération.

D'autre part, les approches portant sur la compatibilité comportementale se focalisent sur la vérification de la cohérence da la séquence d'exécution des opérations entre deux interfaces, sans tenir en compte les aspects structurels des messages envoyés et reçus par ces opérations.

A notre connaissance, il n'a pas des approches existantes permettant d'identifier des incompatibilités mixtes entre l'interface structurelle et comportementale des services Web.

Comme l'interface structurelle des services est décrite en *XML*, les techniques de vérification de la compatibilité structurelle s'appuient sur des algorithmes de comparaison de schémas *XML*. Cependant, les approches pour la vérification de la compatibilité comportementale proposent de modéliser l'interface comportementale des services à l'aide des outils formels tels que les automates et l'algèbre de processus. Puis, d'exploiter les fondements mathématiques de ces théories dans le procédé de comparaison d'interfaces.

2.4 Résolution des incompatibilités

Dans la section précédente, nous avons discuté des tests de compatibilité structurelle et comportementale des services Web. Cette discussion a révélé le fait que la compatibilité entre les interfaces de deux services n'est pas toujours garantie. Ceci est dû, entre autres, à des incompatibilités qui peuvent survenir entre l'interface structurelle et l'interface comportementale des services Web.

Dans cette section, nous discutons de la résolution des incompatibilités qui font que deux interfaces ne sont pas compatibles. Les approches présentées ci-après pour la résolution des

incompatibilités consistent à mettre en place des adaptateurs des interactions entre les services. Ces adaptateurs s'appuient sur un ensemble de règles de mise en correspondance de messages envoyés et reçus. Ces règles ont pour objectif de résoudre les incompatibilités lorsqu'elles existent dans les messages échangés entre deux services.

Dans la section 2.4.1, nous présentons des approches pour la résolution des incompatibilités structurelles, alors que dans la section 2.4.2, nous présentons des approches pour la résolution des incompatibilités comportementales. La section 2.4.3 présente des approches pour la résolution des incompatibilités mixtes. Enfin, La section 2.4.4 fournit une synthèse sur les différentes approches présentées.

2.4.1 Résolution des incompatibilités structurelles

Plusieurs approches ont été proposées pour résoudre les incompatibilités structurelles dans les interactions de services. Notamment, nous distinguons entre des approches à base des techniques de transformation classiques de donnée, et des approches à base des techniques étendues avec l'utilisation des ontologies. Ci-dessous, nous en présentons un panorama.

Techniques de transformation classiques

Fuchs [Fuc04], propose une approche pour la réutilisation de services dans des environnements hétérogènes. L'idée est de permettre à un service de se présenter par différentes interfaces correspondantes chacune à un environnement spécifique. Pour réaliser cela, l'auteur propose d'encapsuler le service avec une couche d'adaptation, dont l'objectif est de transformer selon l'interface originale du service les messages arrivés selon l'interface offerte, et *vice-versa*. La couche d'adaptation s'appuie sur deux notions principales, les *vues de service* et les *groupes de règles*.

La notion de vues est basée sur les techniques de *re-factoring* d'interfaces. Le *re-factoring* [Ref99] est le processus qui consiste à permuter des codes entre des méthodes d'une classe pour améliorer quelques métriques, tels que la structure et l'exécution de classe, sans modifier son interface. Cependant, l'idée ici est de modifier la présentation externe des caractères structurels d'un service, sans toucher sa structure interne. Une *vue* est donc reconnue comme étant une définition *WSDL* dédiée à une audience particulière. Elle décrit l'ensemble d'opérations selon les niveaux abstraits et concrets des spécifications *WSDL*. D'autre part, les *groupes de règles* représentent le coté aspect du processus d'adaptation. Un groupe de règles est associé à une paire d'interface : la nouvelle interface offerte par le service, et l'interface originale. En effet, les règles décrivent la mise en correspondance entre ces paires d'interfaces. A l'exécution, dès qu'un message est reçu, la règle correspondante est activée et

l’adaptation est mise en œuvre.

Dans [KSPBC06], Kongdenfha et coll. présentent une plate-forme basée sur la programmation par aspect pour l'adaptation des interactions de services Web. Au cas où un service d’une composition modifie son interface structurelle ou comportementale, des modifications analogues doivent être accomplies dans les autres services de la composition. Ces modifications ont pour but de restaurer la cohérence avec le service après la modification de ses descriptions. L'approche proposée s'appuie sur la notion de programmation par orientée aspect pour propager de manière flexible les solutions d'adaptation dans les différents services de la composition. Pour chaque type d’incompatibilités, les auteurs associent un modèle de résolution à base de programmation aspect.

Dans [PF04], Ponnekanti et Fox proposent une solution pour résoudre les incompatibilités d’interfaces dans le cadre de la substitution de services. Ils distinguent les incompatibilités suivantes : 1) méthode manquante (les méthodes additionnelles ne causent pas de problème), 2) paramètre d’entrée ou de sortie additionnel ou manquant, 3) domaines de valeurs et cardinalités des paramètres des services non compatibles. Ainsi, ils classifient ces incompatibilités selon les types suivants : incompatibilités structurelles, de valeur, d’encodage, ou sémantiques. Seules les incompatibilités structurelles et de valeur, sont traitées dans ce travail. Les auteurs adoptent les normes de conduite et règles de réutilisation des domaines de nommage et de création de nouveaux domaines de nommage. Si ces normes de conduites sont effectivement suivies, alors les auteurs montrent qu’une compatibilité aux niveaux des valeurs et structures des données implique une compatibilité s´sémantique, que ce soit pour les valeurs échangées, les méthodes, les types ou les champs de types de données. L’approche proposée repose sur plusieurs éléments :

- Des outils d’analyse statique et dynamique (c’est-à-dire durant l’exécution), qui déterminent à partir de règles logiques si un service Web substitué est compatible avec une spécification de processus métier en comparant les paramètres d’entrée/sortie des opérations.

- Des composants semi-automatiquement générés qui résolvent les incompatibilités et établissent les communications avec les services Web substitués.

- Un mécanisme de d’adaptation appelé *multi-option* types qui facilite l’adaptation des paramètres d’entrée/sortie de services en précisant s’ils sont optionnels ou obligatoires, ou lorsqu’ils manquent, s’ils sont remplaçables par des valeurs par défaut.

Aragão et coll. [AF03] proposent une approche à base des règles de style *ECA (Event-Condition-Action)* pour résoudre le problème de conflits dans les services Web. En s'inspirant

des travaux dans le domaine des bases de données, les auteurs proposent une taxonomie d'incompatibilités qui peuvent avoir lieu entre les interfaces de services Web. Cette taxonomie comporte deux axes principaux : les incompatibilités au niveau de descriptions et les incompatibilités au niveau de valeurs. Les incompatibilités au niveau de descriptions se manifestent quand deux services Web décrivent différemment un ou plusieurs éléments dans les messages envoyés et reçus.

Pour résoudre les incompatibilités, l'approche consiste à définir un ensemble de règles d'adaptation de style *ECA*, et de les charger dans le moteur d'exécution *SOAP* du serveur qui héberge le service concernée. Dans une règle *ECA*, un événement se produit quand un message *SOAP* est reçu par le serveur. Un événement déclenche la règle associée. Dès qu'une règle est déclenchée, sa condition est évaluée. Si l'évaluation est positive, l'action d'adaptation correspondante est exécutée. Une telle action consiste à manipuler le corps du message *SOAP* pour le rendre compatible avec ce qui est prévu par le service récepteur.

Dans [TD05], Tolk et Diallo proposent une approche d'ingénierie des données à base de modèles pour adapter les données entre services Web. Leur objectif est de permettre l'interopérabilité des données échangées entre services Web composés, sur la base d'un modèle commun. Ils soulignent les lacunes du langage *XML* concernant la prise en charge de la sémantique attachée aux données, et ils utilisent des méthodes d'ingénierie pour construire manuellement un modèle commun, qui sert de référence aux services Web. La transformation des données vers le modèle commun est effectuée à l'aide d'expressions *XSLT*[11] étendues, qui sont aussi créées manuellement.

Dans [KML06], Kaminski et coll. traitent le problème d'évolution de services limitée à des changements de signatures de ses opérations, et/ou à l'ajout des nouvelles opérations. Ainsi, le service passe d'une version V_1 à une version V_2. Ce changement de version entraine une incohérence dans l'interaction avec ses clients antérieurs, qui continuent à interagir avec le service selon sa version antérieure. Pour restaurer la compatibilité entre le service et ses clients antérieurs, les auteurs présentent une technique à base d'adaptateur.

Cette technique consiste à mettre en place un adaptateur entre deux versions consécutives de services. Par exemple un adaptateur entre les versions V_1 et V_2 d'un service, permet de transformer les messages provenant des clients en termes de l'interface V_1 du service, en messages en termes de l'interface V_2 du service (voir la figure 2.5). De même, il permet de transformer les messages produits par la version V_2 du service, en des messages selon sa version V1 avant de les envoyer aux clients. Au cas où le service passe dans plusieurs

[11] XSLT est un langage de transformation XML

versions, V_1, V_2, ..., V_n, V_{n+1}, les auteurs proposent de déployer une chaine de n adaptateurs. Par exemple, pour qu'un client de la version V_1 du service puisse interagir avec la version V_{n+1}, ses messages envoyés se propagent à travers les n adaptateurs mis en place, tel qu'il est illustré par la figure 2.5.

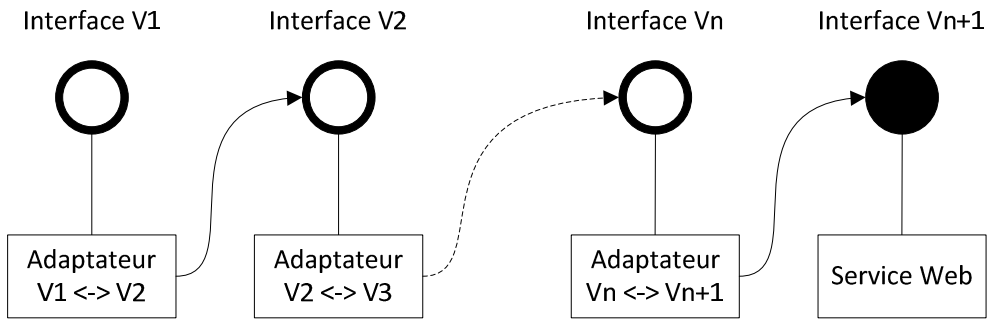

Figure 2.5 Technique de chaines d'adaptateurs

Techniques de transformation étendues avec des ontologies

Dans la communauté du Web sémantique, les ontologies ont été utilisées pour surmonter les hétérogénéités entre les descriptions de services Web afin de favoriser leur découverte, et leur interaction les uns avec les autres. Dans ce contexte, une ontologie peut être perçue comme par un ensemble d'informations qui définissent les concepts utilisés dans un domaine particulier, ainsi que les relations logiques entre eux.

Pour résoudre les incompatibilités entre services, Drong et coll. [DGNT04] s'appuient sur la notion d'ontologie pour décrire non seulement le domaine métier de services, mais aussi les types d'incompatibilités qui peuvent survenir entre ceux-ci, et les règles de résolution correspondantes. En d'autres mots, l'idée est d'associer à chaque type d'incompatibilités, une règle de résolution, puis d'intégrer l'ensemble dans l'ontologie du domaine. Cela permet à l'ontologie d'agir en tant que médiateur d'interaction, capable de détecter et de résoudre les incompatibilités entre les services sans aucune intervention humaine.

Dans [BL04], Bowers et coll. présentent une plate-forme à base d'ontologies pour la résolution d'incompatibilités structurelles dans l'interaction de services. La solution proposée consiste à annoter les descriptions *WSDL* des messages avec des types sémantiques. Ainsi, un message est décrit par un type structurel d'un part et sémantique d'autre part.

La figure 2.6 illustre le principe de la solution de la plate-forme proposée. Au cas où un message d'envoi (P_s) et un autre de réception (P_t) ont des types structurels différents, mais de types sémantiques équivalents, la plate-forme permet de générer une règle de mise en

correspondance entre eux. Celle-ci permet de transformer la structure du message d'envoi en une structure qui correspond à celle du message de réception.

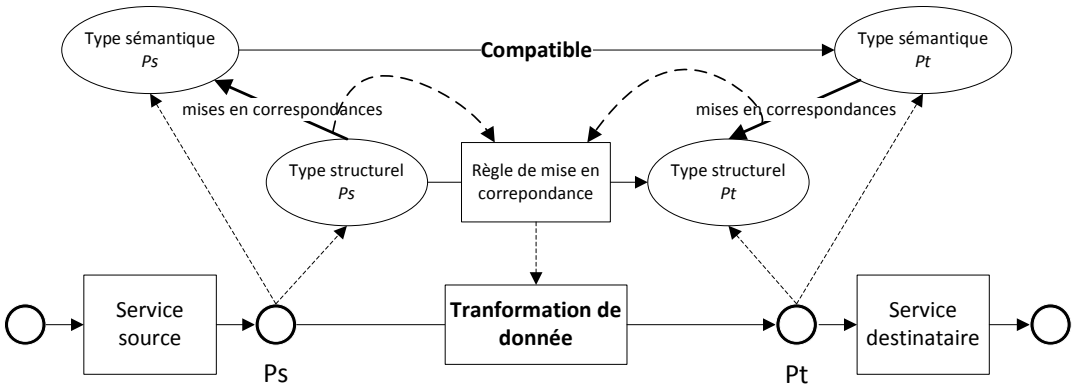

Figure 2.6 Technique de transformation à base d'ontologies

Spencer et coll. [SL04] proposent une technique à base des règles logiques de transformation de données pour l'adaptation des interactions entre des services Web sémantiques. Ils distinguent entre la compatibilité sémantique des données et leur compatibilité structurelle. En effet, la compatibilité sémantique des données peut être vérifiée grâce à l'utilisation de langages de description sémantique, tel que *OWL-S*. Des données mêmes compatibles sur le plan sémantique peuvent être incompatibles sur le plan structurel. Le but de l'approche proposée est de réconcilier les incompatibilités structurelles des données équivalentes sur le plan sémantique. Pour ce faire, les auteurs proposent un compilateur qui permet la génération automatique des règles logiques de transformation.

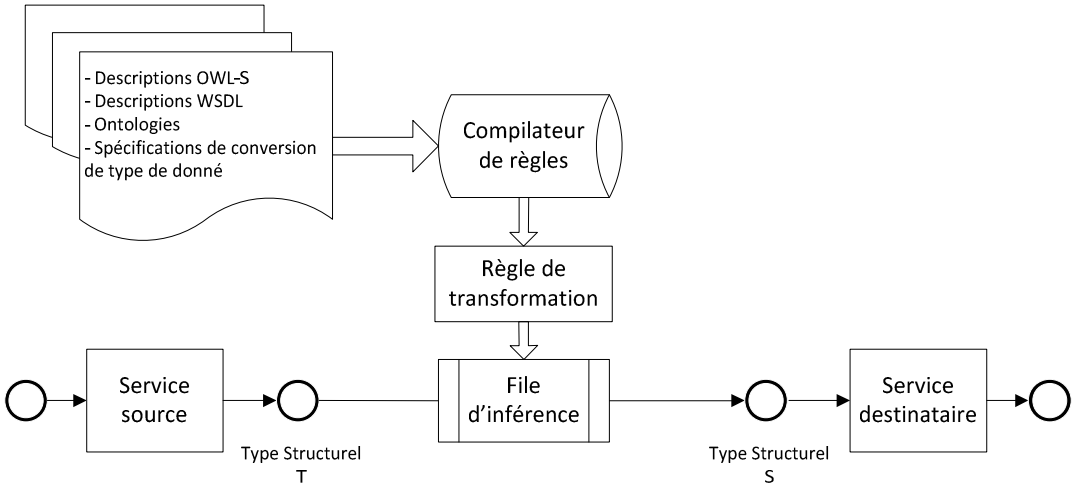

Figure 2.7 Génération des règles logiques de transformation

La figure 2.7 schématise la génération de règles de transformation. Le Compilateur des

règles raisonne aussi bien à partir des descriptions *WSDL* et *OWL-S* fournies par les services, que des spécifications de conversion entre les différents types de données utilisées. Au cas où la transformation structurelle n'est pas possible, le processus d'adaptation échoue. Autrement, les règles de transformation générées (voir la figure) sont déployées dans la File d'inférence qui agit comme médiateur des interactions entre les deux services source et destinataire. Dès qu'un message est reçu de la part du service source, la règle de transformation correspondante est appliquée, et le résultat est envoyé au service destinataire.

Nagarajan et coll. [NVS+06] présentent une architecture de médiation structurelle basée sur l'usage d'ontologies. Selon le même principe que celui présenté dans [BL04], les auteurs proposent d'enrichir les descriptions *WSDL* de messages avec des annotations sémantiques. Ces annotations exprimées en termes de concepts et de propriétés d'une ontologie du domaine permettent de découvrir les correspondances entre les messages d'envoi d'un service et ceux de réception d'un autre. Une fois la correspondance sémantique établie entre les deux messages, des mécanismes d'inférence de l'ontologie génèrent des règles codées en *XSLT* et *XQuery* permettant la transformation de la structure des messages d'envoi en des structures correspondantes à celles de messages de réception.

2.4.2 Résolution des incompatibilités comportementales

L'adaptation comportementale consiste à ajuster la séquence d'envois et de réceptions de messages entre les services. Plusieurs approches ont été proposées selon ce principe. Certaines approches proposent des solutions à base des modèles (automates), d'autres approches proposent des techniques à base d'architectures. Ci-dessous, nous discutons de ces approches.

Approches à bases des modèles

Benatallah et coll. [BCT+06b] présentent une plate-forme qui permet l'analyse, la manipulation et l'adaptation des interfaces comportementales de services, appelées protocoles de services. La contribution principale de leur approche consiste en trois points principaux :

1. Un modèle conceptuel pour la modélisation des protocoles,
2. Une algèbre pour l'analyse et la manipulation des protocoles,
3. Un modèle de développement d'adaptateurs.

Les auteurs considèrent que les langages courants pour la description de protocoles de services *(WS-BPEL, WS-CDL)* ne permettent pas l'automatisation des activités telles que,

l'étude de la compatibilité et l'analyse de conformité des protocoles. Dans ce contexte, la plate-forme propose un modèle conceptuel reposant sur des automates à états finis pour la description de protocoles de services. En se basant sur ce modèle, les auteurs proposent un mécanisme pour l'analyse de la compatibilité des protocoles. Les auteurs proposent deux types d'operateurs. Les premiers prennent en entrée deux protocoles et vérifient s'ils sont complètement ou partiellement compatibles. Les seconds prennent en entrée deux protocoles et retournent leurs scénarios de conversations possibles. Dans le cas où les deux protocoles ne sont pas complètement compatibles, les auteurs proposent de mettre en œuvre un adaptateur. Pour faciliter le développement des adaptateurs, les auteurs proposent une méthode à base de patrons d'adaptation. Ces derniers offrent des directives pour résoudre des incompatibilités dans l'ordre des messages échangés entre les services. Aucune solution pour la génération automatique des adaptateurs n'est proposée.

Nezhad et coll. [NBM+07], développent une méthode pour l'identification et la résolution des incompatibilités entre l'interface structurelle et comportementale de services Web. Sur le plan structurel, les auteurs proposent des adaptateurs permettant d'effectuer la mise en correspondance entre les messages échangés entre les services. Sur le plan comportemental, les auteurs proposent un algorithme qui identifie et adapte les incompatibilités entre les interfaces décrites en automates. Seules les incompatibilités liées aux séquences d'exécution d'opérations ont été traitées. L'algorithme génère un adaptateur permettant de résoudre les interblocages entre les interfaces. Dans ce travail, la détection et la résolution des incompatibilités est semi-automatique. En effet, pour certaines incompatibilités comme par exemple des messages manquants, le concepteur est sollicité pour fournir le message requis. Pour les incompatibilités liées aux séquences d'exécutions d'opérations, un arbre appelé arbre de mise en correspondance est généré automatiquement par l'algorithme. Cet arbre propose des combinaisons de résolution d'une incompatibilité avec les différentes mises en correspondance des messages envoyés et reçus par les services. Cela sert comme un guide pour le concepteur dans son choix de résolution. L'algorithme proposé dans ce travail n'est pas en mesure de détecter et de résoudre des incompatibilités liées à des itérations sur une ou plusieurs opérations, ni celles liées aux choix dans des alternatives.

Ali-Ait-Bachir et Fauvet [ABF07] se concentrent sur le problème de la modification des interfaces comportementales des services. Un service Web peut modifier d'interface suite à une certaine évolution de fonctionnalité. Dans ce cas, les interactions entre ce service et ses clients existants ne peuvent plus aboutir, car ces derniers continuent à interagir avec le service en termes de son ancienne interface. Pour remédier à ce problème, les auteurs proposent de mettre en place un adaptateur des interactions qui intercepte, détecte et réconcilie les conversations incohérentes entre le service et ses consommateurs.

Bussler [Bus03] propose une approche pour l'intégration de systèmes complexes à base de services Web. Il se concentre en particulier sur le problème des incompatibilités entre les interfaces comportementales de servies, appelées spécifications du comportement extérieur de services. Il distingue entre deux types des incompatibilités qui peuvent avoir lieu sur ce niveau. Des incompatibilités au niveau de la structure des messages échangés, et des incompatibilités dans l'ordre d'échanges de ces messages. Pour la résolution des incompatibilités structurelles, l'auteur propose de générer manuellement des règles qui permettent de transformer la structure de messages envoyé par le service émetteur selon la structure de message requis par le service récepteur. En ce qui concerne les incompatibilités comportementales, des règles de synchronisation d'envoi et de réception de messages sont mises en œuvre. Le présent travail touche à la résolution des incompatibilités structurelles et comportementales à la fois, mais la solution proposée reste manuelle. Aucune technique n'a été mise en œuvre pour la génération automatique des règles de médiation, que ce soit sur le plan structurel ou comportemental.

Williams et Coll. [WBC06] proposent une approche de médiation de protocole pour l'adaptation des interactions de services Web sémantique. Ils considèrent le protocole d'un service comme une séquence d'actions communicationnelles comportant : l'envoi d'un message, la réception d'un message, l'envoi d'un accusé de réception, et la réception d'un accusé de réception. Ils distinguent entre un rôle consommateur et un rôle fournisseur de protocoles. Pour décrire les interfaces de services, et résoudre leur incompatibilité, les auteurs ont proposé un langage pour décrire les interfaces d'une façon enrichie sémantiquement à base d'une ontologie. La solution proposée consiste à introduire un composant de médiation dans le système du service consommateur qui permet d'adapter les comportements de ce dernier en fonction des contraintes du service fournisseur.

Approches à base des architectures

Lin et coll. [LGL06] proposent une architecture de médiation pour l'invocation dynamique de services Web dans un environnement hétérogène. L'objectif de l'architecture est de résoudre les incompatibilités liées aux séquences d'échanges de messages. La solution proposée distingue entre deux phases dans l'invocation d'un service : la phase de contrôle et la phase de médiation. La phase de contrôle consiste à comparer l'interface du service consommateur et celle du service fournisseur afin de générer deux catégories des règles. La première catégorie comprend des règles pour la réconciliation des incompatibilités liées aux types de données échangées. La deuxième catégorie comprend des règles qui permettent d'intercepter les appels aux services et d'adapter les séquences de flux de messages envoyés et reçus. Dans la phase de médiation, l'architecture fonctionne comme un médiateur des

interactions entre le service consommateur et le service fournisseur. Au moment de la réception d'un message, le médiateur du comportement vérifie s'il dispose d'une règle transformation de type de données ainsi qu'une règle de transport de message associées au message en question. Si c'est le cas, il s'agit en premier temps d'appliquer la règle de transformation de type de données, puis celle de transport de message qui permet d'envoyer au service destinataire le message ainsi obtenu par la transformation.

Brogi et Popescu [BP06] présentent une architecture pour la génération automatique d'adaptateurs de services Web. Leur approche aborde l'adaptation comportementale et en particulier le cas où l'interface comportementale de services est décrite en *BPEL*. Etant données deux interfaces *BPEL* dont les interactions risquent de ne pas aboutir, l'architecture proposée permet de générer un adaptateur sous forme d'un processus *BPEL*. Une fois mis en œuvre, cet adaptateur permet de résoudre les incompatibilités dans les interactions des deux services concernés. La méthode proposée pour la génération automatique de l'adaptateur consiste en quatre phases principales : 1) transformer les spécifications de services de *BPEL* en *YAWL*[12][Hof05] , 2) examiner les interfaces *YAWL* et générer un adaptateur sous forme d'un processus *YAWL*, 3) vérifier certaines caractéristiques de problèmes liés aux interactions de services (inter-blocages, etc.) grâce aux techniques abstraits offerts par *YAWL*, 4) transformer le processus d'adaptation *YAWL* en *BPEL* et déployer l'adaptateur.

Dans [HCM+05], les auteurs utilisent un moteur de chorégraphie reposant sur des médiateurs afin de réconcilier les séquences d'interactions des services Web et de leurs clients par l'intermédiaire de l'architecture *WSMX*. Ces médiateurs sont chargés des tâches telles que le groupement de plusieurs messages en un seul, le changement de l'ordre des messages, ou la suppression de messages inutiles. Les auteurs précisent que ces médiateurs fonctionnent grâce à un travail manuel effectué lors de la phase de conception du processus métier. Ce travail consiste à créer des règles, qui permettent d'identifier les équivalences entre les différentes chorégraphies. Ces règles, une fois stockées, sont utilisées durant l'exécution par le moteur de chorégraphie afin de réconcilier les différentes gestions des séquences de messages. En pratique, l'architecture *WSMX* elle-même est implantée comme un service Web, qui fournit les fonctionnalités proposées par l'intermédiaire de ces opérations.

Dans [CD05], les auteurs modélisent les contraintes de communication par un ensemble de règles logiques. Un médiateur de processus s'occupe de résoudre les incompatibilités dans les interactions entre les services en utilisant ces règles, tout en se positionnant entre le service consommateur et le service fournisseur afin d'intercepter les messages échangés et de les adapter. Il utilise un composant appelé interpréteur de chorégraphie, qui permet d'exécuter un

[12] Langage abstrait pour la représentation des interactions et des comportements des services à base de réseaux de petri

service Web en créant les messages requis par ce dernier sur la base de sa description. Il garde en mémoire l'´etat d'avancement des interactions entamées avec le service Web, en mémorisant les appels exécutés par un composant appelé invocateur.

Pires et coll. [PBM03] identifient le problème des incompatibilités structurelle et comportementale de services Web. Ils présentent une approche multi niveau qui repose sur l'utilisation de *services médiateurs*, qui sont des services virtuels fournissant une interface homogène permettant d'accéder à des services Web fournissant la même fonctionnalité ; mais adoptant des interfaces qui peuvent être différentes. En particulier, ces adaptateurs sont chargés de transformer les messages provenant en termes de l'interface virtuels en termes d'interfaces actuels de services, et *vice-versa* pour les messages envoyés.

2.4.3 Résolution des incompatibilités mixtes

Nous présentons maintenant des approches qui portent sur la résolution des incompatibilités mixtes de l'interface structurelle et comportementale de services Web.

Benatallah et coll. [BCG+05] présente une méthode pour le développement des adaptateurs de services Web à base de la technologie *BPEL*. Les auteurs caractérisent des patrons des incompatibilités qui peuvent avoir lieu entre les interfaces des services. Les incompatibilités caractérisées sont de type structurel, comportemental, et mixte. En ce qui concerne les incompatibilités mixtes, les patrons traités sont ceux liés au regroupement de plusieurs messages en un seul, et la séparation d'un message en plusieurs.

Après avoir caractérisé les patrons d'incompatibilités, les auteurs proposent pour chacun de ceux-ci un patron de résolution correspondant. Les patrons de résolution sont décrits en pseudo code sous forme d'un fragment de processus *BPEL*. Ces patrons servent comme référence pendant le développement d'adaptateurs. Notamment, lors de la mise en œuvre d'un adaptateur, le développeur est chargé de repérer manuellement les incompatibilités entre les interfaces de services, puis de mettre en œuvre le patron de résolution correspondant. La mise en œuvre de ce patron est manuelle : le développeur doit être en mesure de traduire le pseudo code proposé par le patron en un code réel exprimé en `BPEL`. Par exemple, si l'interface d'un service décrit une opération de réception de message des informations civiles (nom, prénom, adresse) en une seule structure que le consommateur envoie ces messages séparément, l'opérateur dit de *fusion* est utilisé. Celui-ci décrit en pseudo code le processus de résolution qui consiste à recevoir plusieurs messages provenant de différentes opérations d'envoi du consommateur, puis il crée une structure de message unique telle que définie dans l'interface du service. Une fois que la structure de message est conforme à la structure du message défini dans l'interface fournie du service, le patron envoie ce nouveau message au service. Le

développeur est chargé de traduire ce processus en *BPEL*.

Dans ce travail, la détection d'incompatibilités et la génération d'adaptateurs sont manuelles. Les patrons d'incompatibilités et de résolution proposés servent à guider les développeurs lors du développement d'adaptateurs. Aucune méthode n'a été proposée pour la génération automatique d'adaptateurs. Ajoutons que les auteurs n'étudient pas la complétude de leurs patrons d'incompatibilités et de résolution. Ils se limitent à présenter chaque patron à part, sans souligner s'il est possible de composer deux ou plusieurs opérateurs de résolution au sein d'un adaptateur.

Dumas et coll. [DSW06] traitent le problème d'incompatibilités mixtes entre les interfaces fournie et requise d'un service. Les auteurs modélisent une interface comme une séquence ordonnée d'actions d'envoi et de réception des messages. Ainsi, ils développent une algèbre de transformation contenant six opérateurs permettant d'établir des correspondances, à partir de l'interface fournie d'un service vers son interface requise. Ces six opérateurs permettent d'adapter une action d'une interface à l'autre, de fusionner plusieurs actions en une seule, de fractionner une action en plusieurs, ou bien d'agréger plusieurs messages provenant d'une même action en un seul, et inversement. Enfin, un opérateur permet supprimer un message non requis. Un langage visuel de représentation des transformations assiste les concepteurs de composition dans le processus d'adaptation des interfaces de services Web. L'implantation de la proposition comprend un outil visuel de transformation d'interfaces, ainsi qu'un moteur de composition de services supportant une adaptation qui repose sur l'algèbre et la notation visuelle proposées. Le déploiement d'une transformation d'interface dans le moteur de composition a pour résultat un nouveau point d'accès de service qui correspond aux exigences requises par l'interface avec laquelle il communique. Ce point d'accès transmet les messages au service adapté, tout en effectuant les transformations requises à la volée. Dans ce travail, la détection d'incompatibilités et la génération d'adaptateurs sont manuelles. C'est le développeur qui s'occupe de l'identification des incompatibilités entre les interfaces requise et fournie, ainsi de la mise en œuvre des opérateurs d'adaptation correspondants.

2.4.4 Synthèse

Dans cette section, nous proposons une synthèse des approches que nous venons de présenter. Cette synthèse permet de mettre en évidence non seulement les apports de ces approches mais aussi leurs limites. Dans cette synthèse, nous intéressons aux types des incompatibilités traités, aux techniques utilisées pour la résolution des incompatibilités, et à l'automatisation de la résolution des incompatibilités (automatique/manuelle).

Type des incompatibilités traitées

Nous avons classé les approches présentées selon le type d'incompatibilités traitées, à savoir des approches pour la résolution des incompatibilités structurelles, comportementales, et mixtes.

Les approches pour la résolution des incompatibilités structurelles se concentrent sur la réconciliation de différences structurelles au niveau de messages échangés entre les services. Le principe de ces approches consiste à mettre en place des solutions d'adaptation permettant de transformer la structure des messages envoyés par un service en une structure qui est conforme à celle des messages reçus.

Type / Approche		structurel	comportemental	mixte
Résolution des incompatibilités	[ABF07]		X	
	[AF03]	X	X	
	[BCG+05]		X	X
	[BCT+06]		X	
	[BL04]	X		
	[BP06] [Bus03]		X	
	[CD05]	X	X	
	[DGNT04]	X		
	[DSCW06]		X	X
	[Fuc04]	X		
	[KML06]	X	X	
	[KSPBC06]	X	X	
	[NBM+07]	X	X	
	[NVS+06]	X		
	[PBM03]	X	X	
	[SL04]	X		
	[WBC06]		X	

Tableau 2.1 Synthèse selon le type des incompatibilités traitées

Les approches pour la résolution des incompatibilités comportementales se focalisent sur la réconciliation des incompatibilités liées aux séquences d'exécutions des opérations dans les interfaces de services. Le principe des solutions proposées consiste à mettre en place une solution d'adaptation permettant de synchroniser l'exécution des opérations correspondantes entre deux services qui interagit l'un avec l'autre.

Les approches pour la résolution des incompatibilités mixtes se focalisent sur la réconciliation des incompatibilités liées à la fois aux aspects structurels et comportementaux. Par exemple, des multiples exécutions d'une opération d'envoi de message *versus* une exécution unique d'une opération de réception de message décrite, etc.

Dans le tableau 2.1, les approches [BL04, DGNT04, SL04, Fuc04, NVS+06] proposent des méthodes pour la résolution des incompatibilités structurelles, sans prendre en compte les aspects comportementaux. Cependant, [Bus03, BCT+06, WBC06, ABF07, NBM+07, BP06] proposent des solutions pour la résolution des incompatibilités comportementales sans prendre en compte les aspects structurels.

Les approches [CD05, KSPBC06, PBM03, KML06, AF03] proposent des solutions pour la résolution des incompatibilités aussi bien structurelles que comportementales.

Le tableau 2.1 montre que très peu d'approches on été proposées pour traiter les incompatibilités mixtes. Seules les approches [BCG+05] et [DSCW06] traitent ce type des incompatibilités.

Techniques utilisées

Les approches pour la résolution des incompatibilités structurelles s'appuient sur deux catégories des techniques de résolution.

La première catégorie est basée sur des techniques de transformation classiques des données *XML*. Les fonctions de correspondance entre les différents éléments de deux descriptions *WSDL* sont établies de manière manuelle, car il n'est pas trivial d'automatiser le processus d'établissement des correspondances entre deux schémas *XML* sans posséder des informations sémantiques sur les éléments de schémas. Une solution particulière, comme dans les travaux de [PF04] consiste à supposer que certaines règles sont respectées au niveau des méthodes de nommage. Dans ce cas particulier, les hétérogénéités liées aux noms d'opérations et de messages sont résolues. Cependant ce genre de solution reste difficilement applicable car il est difficile que les fournisseurs de services adhèrent à un vocabulaire commun.

La deuxième catégorie des approches s'appuient sur la notion d'ontologie pour faciliter et automatiser la résolution des incompatibilités. Etant donné que les descriptions *WSDL* des services sont annotées avec des informations sémantiques, le principe de ces approches consiste à exploiter ces informations afin de générer automatiquement les règles de mise en correspondance entre les interfaces *WSDL* de services.

Technique / Approche		Classique	Basée sur les ontologies
Résolution des incompatibilités	[AF03]	X	
	[BL04]		X
	[DGNT04]		X
	[Fuc04]	X	
	[KML06]	X	
	[KSPBC06]	X	
	[NVS+06]		X
	[PF04]	X	
	[SL04]		X
	[TD05]	X	

Tableau 2.2 Synthèse selon les techniques de résolution des incompatibilités structurelles

Comme indiqué dans le tableau 2.2, les approches [Fuc03, KSPBC06, PF04, AF03, TD05, KML06] s'appuient sur des techniques de transformation classique des données *XML* sans résoudre des incompatibilités liées à la sémantique des données. Cependant, les approches [DGNT04, BL04, SL04, NVS+06] s'appuient sur les ontologies pour résoudre aussi bien les incompatibilités liées au nommage des éléments d'interfaces *WSDL*, que pour générer automatiquement des règles de mise en correspondance structurelle entre les interfaces.

Nous avons classé en deux catégories les approches pour la résolution des incompatibilités comportementales. La première catégorie comprend des approches basées sur des modèles formelles. La deuxième catégorie comprend des approches basées sur des architectures qui décrivent informellement le procédé de la mise en œuvre des adaptateurs.

Les approches [BCT+06b, NBM+07, ABF07, Bus03, WBC06] proposent de modéliser les interfaces de services par des automates. Une fois les interfaces modélisées, l'idée consiste à exploiter la théorie formelle de ce modèle, telle que les propriétés de simulation et bissimulation, afin de détecter et résoudre les incompatibilités. En particulier, [BCT+6] s'appuie sur des opérateurs algébriques définis sur les systèmes de transitions, tels que l'union, l'intersection, l'addition, et la soustraction de deux automates dans sa méthode proposée pour la construction des adaptateurs. Dans sa méthode de résolution des incompatibilités, [NBM+07] s'appuie sur des automates afin de générer automatiquement un arbre de mise en correspondances entre deux automates, proposant une combinaison de résolution d'incompatibilités identifiées entre les automates. [ABF07] étend sur la propriété de bissimlation entre deux automates afin de détecter et résoudre des incompatibilités comportementales entre des services. [Bus03] et [WBC06] s'appuient sur des automates pour

synchroniser les interactions entre deux services en éliminant les interblocages.

Technique / Approche		Basée sur les modèles	Basée sur les architectures
Résolution des incompatibilités	[ABF07]	X	
	[BCT+06b]	X	
	[BP06]		X
	[Bus03]	X	
	[CD05]		X
	[HCM+05]		X
	[LGL06]		X
	[NBM+07]	X	
	[PBM03]		X
	[WBC06]	X	

Tableau 2.3 Synthèse selon les techniques de résolution des incompatibilités comportementales

Les approches à base d'architectures proposent des méthodes qui agissent directement sur les interfaces de services telles qu'elles sont fournies par ceux-ci. Les architectures sont utilisées pour décrire les différentes étapes proposées dans le processus de résolution, ainsi que les différents composants qui font partie de la solution proposée. Notamment, l'architecture de médiation proposée par Lin et Coll. [LGL06] décrit les deux phases de la méthode proposée, à savoir la phase de contrôle et la phase de médiation, ainsi que fonctionnement des règles de médiation proposées qui interceptent et adaptent les messages échangés entre les services. De même, l'architecture proposée par Brogi et Popescu [BP06] décrit les différents composants proposés pour la génération automatiques des adaptateurs de services Web à base de la technologie BPEL. Les approches [PBM03, HCM+05, CD05] proposent des architectures qui s'appuient sur l'utilisation des médiateurs à base de règles de résolution des incompatibilités lors des interactions entre les services. Ces architectures régissent le fonctionnement des différents ensembles de règles proposées : structurelles et comportementales. Le tableau 2.3 résume cette discussion.

Génération automatique des adaptateurs

Des approches proposent des mécanismes permettant la génération automatique des adaptateurs [Fuc04, TD05, PF04, AF03, KSPBC06, KML06, DSW06, KML06, NBM+07, Bus03, LGL06, HCM+05, CD05, PBM03]. D'autres proposent des directives et laisse la tâche de génération de code à un développeur humain [PF04, NVS+06, BCT+06b, ABF07, DGNT04, NVS+06, WBC06, BP06]. Tableaux 2.4 dresse un bilan de ces approches.

Automatisation / Approche		Automatique	manuel
Résolution des incompatibilités	[ABF07]	X	
	[AF03]		X
	[BCT+06b]	X	
	[BL04]	X	
	[BP06]	X	
	[Bus03]		X
	[CD05]		X
	[DGNT04]	X	
	[DSW06] [Fuc04]		X
	[HCM+05]		X
	[KSPBC06] [KML06]		X
	[LGL06] [NBM+07]		X
	[NVS+06]	X	
	[PF04] [PBM03]		X
	[SL04]	X	
	[TD05]		X
	[SL04] [WBC06]	X	

Tableau 2.4 Synthèse selon l'automatisation de la résolution des incompatibilités

2.5 Conclusion

Dans ce chapitre, nous avons présenté les différentes approches qui traitent le problème des incompatibilités des interfaces entre deux services qui interagit l'un avec l'autre. Les approches proposées portent sur la détection et la résolution des incompatibilités structurelles, comportementales et mixtes des interfaces des services.

La synthèse que nous avons établie sur les approches existantes a permis révéler les limites suivantes :

- L'absence de méthodes pour la détection des incompatibilités mixtes,
- L'absence de méthodes pour la génération automatique d'adaptateurs permettant de résoudre des incompatibilités mixtes,
- L'absence d'un environnement intégré permettant de combiner la détection et la résolution des incompatibilités dans un cadre opérationnel.

Dans la deuxième partie de cette thèse, nous proposons un canevas pour l'adaptation des interactions des services par génération automatique des adaptateurs. Dans ce canevas, l'adaptation porte sur la détection et la résolution des incompatibilités mixtes entre les interfaces de services.

.

Deuxième partie

Proposition d’un canevas pour l’adaptation d’interactions des services par génération semi-automatique d’adaptateurs

CHAPITRE 3

Modélisation de l’interface des services

Sommaire

3.1 Introduction

Dans la première partie de ce manuscrit, nous avons positionné le problème d’adaptation par rapport à celui de la substitution de services. Notamment, nous avons mis en évidence le rôle primordial de l’adaptation pour surmonter les problèmes des incompatibilités produites entre l’interface du service consommateur et celle du service fournisseur substitut.

Dans cette deuxième partie du manuscrit, nous introduisons la proposition d’un canevas pour l’adaptation des interactions entre deux services par génération automatique d’un adaptateur. Alors que notre proposition est motivée par un scénario de substitution de services, elle est valable pour tout scénario de sélection dynamique de service : que ça soit dans un cadre de substitution ou bien de composition de services.

Etant donné deux services qui souhaitent interagir l’un avec l’autre, l’objectif principal du canevas proposé est, d’une part de détecter les incompatibilités entre leurs interfaces et, d’autre part de générer un adaptateur permettant de résoudre ces incompatibilités lors des interactions entre les deux services.

Dans la figure 3.1, l’interface du service consommateur est conçue de manière adaptée à

l'interface du service Fournisseur original. La substitution du Fournisseur original par un service substitut, provoque des incompatibilités entre l'interface du service Consommateur et celle du service Fournisseur substitut. Ainsi, les interactions entre les deux services ne peuvent pas aboutir. Pour surmonter ce problème, le composant Adaptateur a été mis en place. Son rôle est de transformer les interactions provenant du service Consommateur selon l'interface du service Fournisseur original en des interactions selon l'interface du service Fournisseur substitut, et *vice-versa*.

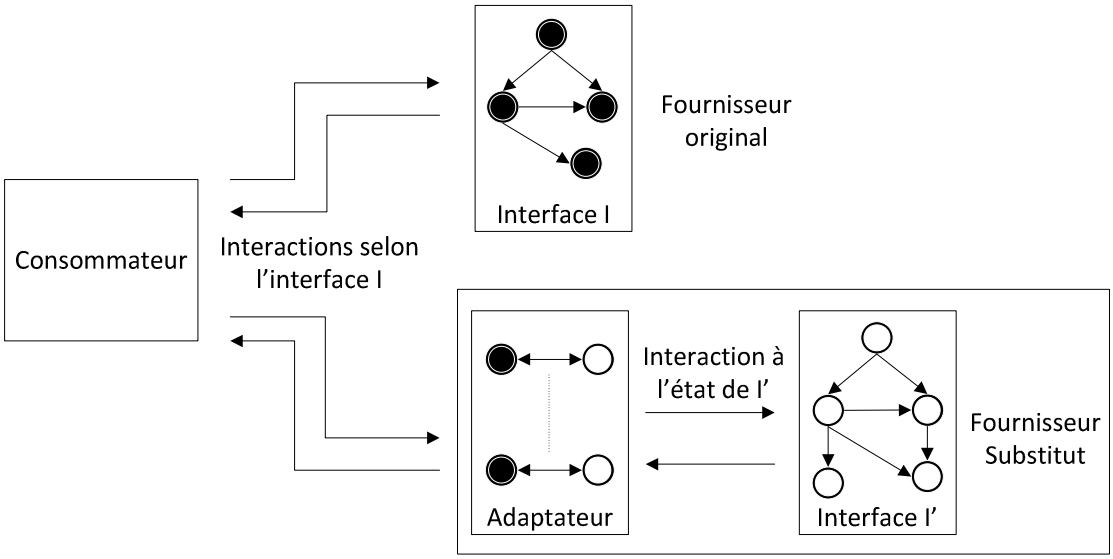

Figure 3.1 Adaptation des interactions entre un consommateur et un fournisseur substitut

La figure 3.2 schématise le principe général de notre proposition pour la détection des incompatibilités et la génération automatique des adaptateurs. D'abord (1), il s'agit de modéliser l'interface structurelle et comportementale *(décrites respectivement en WSDL et WSCI)* de chacun des deux services Consommateur et Fournisseur à l'aide des automates déterministes en nombre fini d'états. Ensuite (2), de mener des tests de compatibilité entre les automates décrivant les interfaces des deux services afin d'identifier leurs différences qui peuvent engendrer des incompatibilités à l'exécution. Enfin (3), de générer un adaptateur sur la base des différences détectées afin de résoudre les incompatibilités dans les interactions des deux services.

Alors que le présent chapitre discute de la modélisation en automates des interfaces structurelle et comportementale des services, le chapitre suivant présente notre méthode pour la détection des incompatibilités et la génération automatique des adaptateurs.

Ce chapitre est organisé comme suit. La section 3.2 détaille la modélisation à base

d’automates des interfaces structurelle et comportementale des services. La section 3.3 introduit des propriétés sur la modélisation proposée qui permettent de vérifier formellement si un service consommateur peut interagir correctement avec un fournisseur donné, ou bien si un service donné peut remplacer un autre sans recourir à une adaptation. Enfin, la section 3.4 conclut le chapitre.

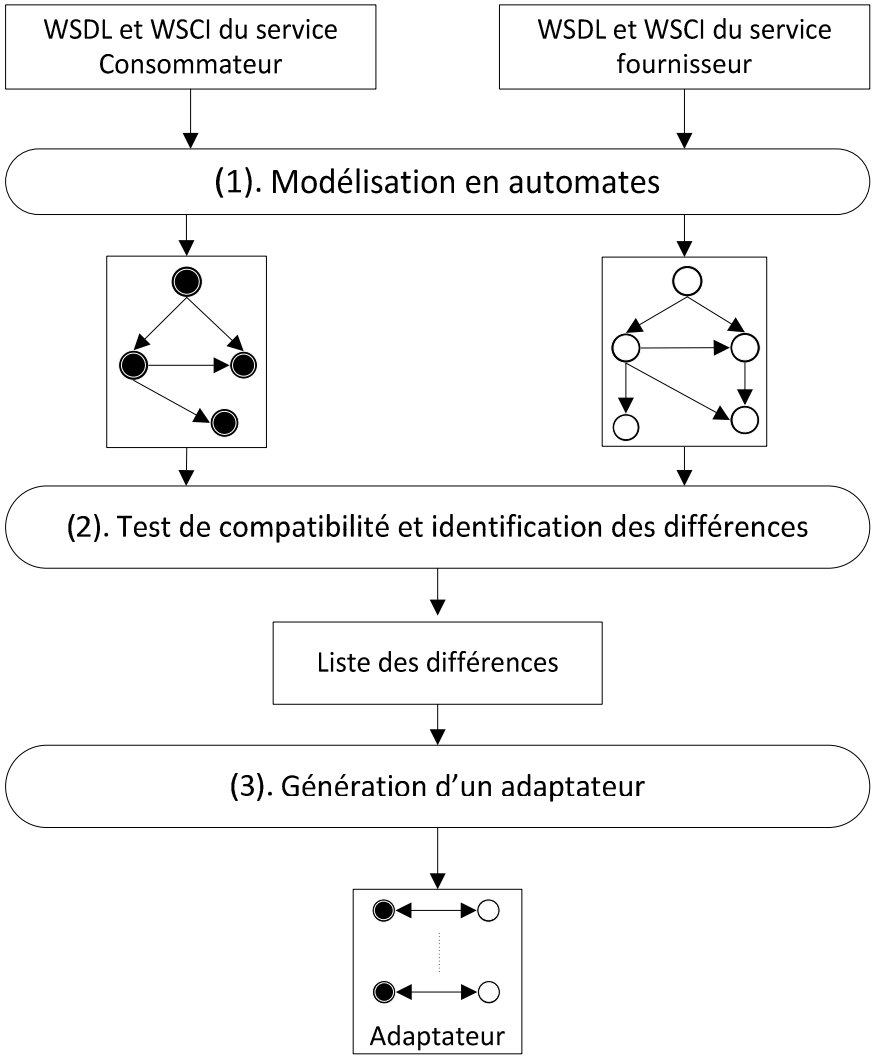

Figure 3.2 Principe générale de notre proposition

3.2 Interfaces de services

Alors que l’interface structurelle d’un service décrit ses opérations offertes, son interface comportementale décrit la séquence d’exécution de ces opérations. Dans cette section, nous modélisons l’interface structurelle et comportementale d’un service Web par un système de transitions étiquetées (*Labelled Transistion Systems*, *LTS*). Cette modélisation est présentée dans les sections 3.2.1 et 3.2.2. La section 3.2.2 introduit un jeu des opérateurs sur les

interfaces de services, utilisées dans la suite de la thèse lors de la détection et de la résolution des incompatibilités.

3.2.1 Interface structurelle

L'interface structurelle décrit l'ensemble des opérations offertes par un service Web. Une opération est soit :

- un envoi du message m : noté !m
- une réception du message m : notée ?m

En *WSDL*, une opération est caractérisée par une polarité d'envoi/réception et d'un message envoyé/reçu. Ce dernier est caractérisé par un nom et un type structurel décrit en un schéma *XML*.

Nous désignons par polarity le type abstrait[13] qui modélise la polarité d'envoi ou de réception des messages, par message le type abstrait qui modélise les messages envoyés ou reçus, et par operation le type abstrait qui modélise les opérations offertes par les services Web. Les types polarity, message, et operation sont décrits ci-dessous :

Type enum polarity : { ?, !}
{polarity est le type énuméré qui consiste en deux symboles ? et !}

type message ⟨
name : string {Nom du message}
type : XML {Le type structurel du message décrit par un schéma XML}
⟩

type operation ⟨
m : message {le message envoyé or reçu par l'opération}
p : polarity {La polarité (envoi/réception) du message associé à l'opération}
⟩

Ci après, nous spécifions un jeu d'opérations sur ces types. Ce jeu d'opérations est utilisé dans la suite de ce travail pendant la comparaison des interfaces, la détection des incompatibilités, et la génération des adaptateurs.

[13] http://fr.wikipedia.org/wiki/Type_abstrait

Message : operation ⟶ message

{Message (op) est le message envoyé ou reçu par l'opération op.}

Polarity : operation ⟶ polarity

{Polarity(op) est la polarité de l'opération op. Le symbole ? (respectivement !) est retourné si op est une opération de réception (respectivement d'envoi). P étant une polarité, P = ? ⇔ $\overline{P}$ = ! }

Name: message ⟶ string

{Name(m) est le nom du message m décrit par une chaine de caractère. Ce nom est donné dans la description WSDL du service.}

Type: message ⟶ XML

{Type(m) est le type structurel du message m décrit par un schéma XML. Ce schéma est donné dans la description WSDL du service.}

Dans ce qui suit, nous définissons la notion d'équivalence entre deux types structurels décrivant deux messages différents. Dans cette définition, nous nous appuyons sur un algorithme proposé par Zhang et coll. [ZYGW06].

Etant donné deux types structurels différents l'algorithme permet de retourner le degré de leur équivalence. Dans le procédé de comparaison, les auteurs s'appuient sur des algorithmes tels que *NGram*, *SemanticMatch*, *ChekAbbreviations* afin de calculer la similarité linguistique entre les noms des éléments de messages. Le degré de similarité linguistique finale (entre 0 et 1) correspond à la moyenne des degrés retournés par chacun des trois algorithmes. Au sujet de la comparaison des types de valeurs des éléments de messages, l'algorithme considère deux types de valeur comme compatibles s'ils sont identiques, semi-compatible s'ils peuvent être adaptés l'un par rapport à l'autre, c'est le cas de int et float, ou incompatible si aucune adaptation n'est possible entre eux, c'est le cas de int et string. En ce qui concerne les types complexes, l'algorithme propose de comparer chaque sous-élément du premier avec chaque sous-élément du second. A chaque paire ainsi constituée est associée une valeur de similarité. Les paires ayant le score maximal sont considérés correspondantes.

Nous illustrons l'algorithme de Zhang et coll. à travers un exemple. Soit Item et Product deux messages de types structurels respectivement Type(Item) = tns :Item et Type(Product) = tns :Product (voir tableau 3.1). Le type tns :Item est composé des deux éléments : quantity et product, alors que le type tns:Product est composé des deux éléments : amount et productName. L'algorithme de Zhang et coll. appliqué sur ces deux types, retourne les résultats donnés dans le tableau 3.2.

<complexType name= “tns:Item”> <all> <element name=“quantity” type=“int”> <elemnt name = “product” type = “string”> </all> </Complex Type>	<complexType name= “tns:Product’> <all> <element name=“amount” type=“int”> <elemnt name = “productName” type = “string”> </all> </Complex Type>

Tableau 3.1 Types structurels des deux messages Item et Product

tns :product / tns :Item	ProductName : str	Amount : int
Quantity : int	0	0.8
Product : str	0.9	0

Tableau 3.2 Exemple de comparaison des types complexes

Dans le tableau 3.2, l’algorithme fait correspondre l’élément quantity (respectivement Product) du type tns :Item à l’élément Amount à un degré 0.8 (respectivement à l’élément ProductName à un degré 0.9) du type tns :Product. La moyenne des résultats donnés indique que le degré d’équivalence entre les deux types structurels tns :Item et tns :Product est de à 0.85. D’après une étude expérimentale fournie par ce travail, un degré qui s’élève à 0.85 entre deux types structurels indique une similarité importante entre ceux-ci.

Dans notre canevas, nous considérons que deux types T_1 et T_2 sont équivalents, ce qui noté $\mathbf{T_1 \equiv T_2}$, si leur degré d’équivalence donné par l’algorithme de Zhang est supérieur ou égale à 0.8. Ce choix est justifié par une étude expérimentale que nous avons menée sur un ensemble des types équivalents, pour lesquels l’algorithme de Zhang a donné des résultats qui varient entre 0.8 et 1.

3.2.2 Interface comportementale

Plusieurs travaux ont utilisé les automates pour modéliser le comportement d’un objet, d’un système informatique, ou bien d’un processus d’interaction entre deux ou plusieurs systèmes [BSBM04, BCT06a, MPC01]. Nous nous sommes inspirés de ces travaux pour modéliser l’interface comportementale des services à l’aide d’automates.

Un automate déterministe en nombre fini d’états est une machine abstraite permettant de représenter l’évolution d’un système par un graphe orienté qui comporte des états reliés par des transitions étiquetées par des actions [HMU00].

Un état d'un automate représente un état du système modélisé. Un automate comprend un état initial, un ensemble des états intermédiaires, et un ou plusieurs états finaux.

Une transition d'un automate relie deux états et modélise l'occurrence d'un événement qui entraine le passage de l'automate de l'état source de la transition à l'état cible. Une transition ne relie pas nécessairement des états distincts. Pour certaines actions, l'état cible peut être l'état source lui-même (voir dans la figure 3.3 la transition étiquetée par ?(commander un article)). A un instant donné, l'évolution du système se traduit par le passage d'un état à un autre en franchissant l'une des transitions qui relie les deux états.

Dans notre modélisation des interfaces des services Web, les états de l'automate représentent l'ensemble des états dans lequel un service peut passer lors de ses interactions avec un consommateur. Les transitions correspondent aux différentes actions *communicationnelles* qu'un service peut réaliser pendant ses interactions avec son consommateur. Ces actions sont en réalité les opérations d'envoi de réception de message du service. Nous distinguons deux types de transitions, à savoir :

- Des transitions qui correspondent à une opération de réception de message. Elles sont étiquetées par le message reçu, précédé du symbole ?. Dans la figure 3.3, la transition étiquetée par ?(Commander un article) correspond à l'opération de réception d'un message envoyé par le consommateur pour ajouter un article à sa commande.

- Des transitions qui correspondent à une opération d'envoi de message. Elles sont étiquetées par le message envoyé par l'opération, précédé par du symbole !. Dans la figure 3.3, la transition étiquetée par !(Devis de commande) correspond à l'opération d'envoi d'un message au consommateur contenant sa demande de devis.

L'état initial d'un automate décrivant un service consiste en l'attente d'une demande de traitement envoyée par le consommateur. Quand une action d'envoi ou de réception de message est exécutée, la transition associée est déclenchée, et le service change d'état (*il passe de l'état source à l'état cible de la transition*). Dans un souci de clarification, les états de cet exemple sont nommés, alors que nous leur donnons des numéros dans les exemples suivants.

La figure 3. 3 illustre l'exemple d'un automate modélisant l'interface comportementale d'un service de vente en ligne. Pour traiter une commande de n lignes, le service exécute n fois une opération lui permettant de recevoir une ligne de commande. Chaque exécution correspond au déclenchement de la transition étiquetée par ?(Ligne de commande). Comme l'état cible de cette transition est l'état source lui-même, le service reste toujours dans le

même état Attente d'une commande jusqu'à la réception d'un message de validation de panier.

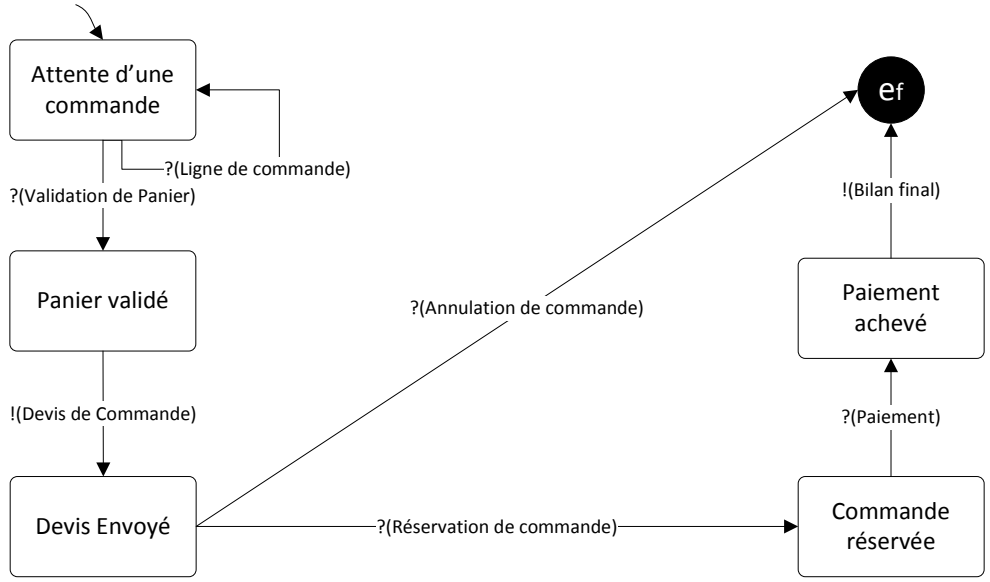

Figure 3.3 Modélisation en automate de l'interface comportementale d'un service

La réception du message de validation de panier permet de déclencher la transition étiquetée par ?(Validation de panier), ensuite le service passe à l'état de Panier validé. A ce stade, le service est dans un état qui lui permet de calculer le devis de la commande et de l'envoyer au consommateur. L'exécution de cette opération déclenche la transition étiquetée par !(Devis de commande), ce qui permet au service de passer dans l'état Devis envoyé. Ici, le service est en état d'attente de deux alternatives :

- La demande d'annulation du consommateur qui se traduit par le déclenchement de la transition étiquetée par ?(Annulation de commande). Ce qui entraine le passage du service dans l'état final.

- L'accord du consommateur qui se traduit par le déclenchement de la transition étiquetée par ?(Réservation de commande), ensuite le service passe dans l'état Commande réservée.

Une fois la commande réservée, le service s'attend à la réception du message de paiement ?(Paiement) de la part du consommateur avant de lui envoyer le bilan final de la commande !(Bilan final). Enfin, le service passe dans l'état final, et son exécution se termine.

En nous appuyant sur la définition classique d'automate, nous fournissons ci-après la définition d'une interface de service modélisée par un automate :

Définition 3.2.1 (Interface d'un service)

Une interface d'un service modélisée par un automate déterministe en nombre fini d'état est un tuple <E, e_0, E_f, M, T> tel que :

1. E est l'ensemble d'états d'un service,
2. $e_0 \in E$, est l'état initial du service,
3. $E_f \subseteq E$ est l'ensemble d'états finaux du service,
4. M est l'ensemble des messages envoyés et reçus par le service.
5. $T \subseteq E \times \{ ?, !\} \times Message \times E$ est l'ensemble des transitions

Une transition notée <e, ?a, e'>, qui s'écrit également $q \xrightarrow{?a} q'$, modélise le passage du service de l'état e à l'état e' suite à la réception du message a. Dans l'exemple de la figure 3.3, la transition <Attente de commande, ?(Validation de panier), Panier validé> traduit le passage de l'automate de l'état Attente de commande à l'état Panier validé suite à la réception du message Validation de panier.

Une transition notée <e, !b, e'>, qui s'écrit également $q \xrightarrow{!b} q'$, modélise le passage du service de l'état e à l'état e' suite à l'envoi du message b. Dans l'exemple de la figure 3.3, la transition < Panier validé, ?(Devis de commande), Devis envoyé> traduit le passage de l'automate de l'état Panier validé à l'état Devis envoyé suite à l'envoi du message Devis de commande.

Une trace est la séquence des transitions [<e_i, a_i, e_{i+1}> , <e_{i+1}, a_{i+1}, e_{i+2}> , … , <e_{n-1}, a_{n-1}, e_n>] permettant d'atteindre l'état e_n à partir de l'état e_i. Il s'agit de franchir la transition <e_i, a_i, e_{i+1}> pour atteindre l'état e_{i+1} et ainsi de suite jusqu'au franchissement de la transition <e_{n-1} , a_{n-1} , e_n>. Dans l'exemple de la figure 3.3, [<Attente de commande, ?(Validation de Panier), Panier validé> , <Panier validé, ?(Devis de commande), Devis envoyé>] est la trace permettant d'atteindre l'état Devis envoyé à partir de l'état Attente de commande.

Une trace acceptée est une *trace* permettant d'atteindre l'état final de l'automate à partir de son état initial. Dans la figure 3.3, la trace [<Attente de commande, ?(Ligne de commande), Attente de commande > , <Réception de commande, ?(Validation de Panier), Panier validé> , <Panier validé, !(Devis de commande), Devis envoyé> , <Devis envoyé, ?(Annulation de commande), e_{final}>] est une trace acceptée.

Un état e' est dit **atteignable** depuis l'état e (noté $e \xrightarrow{*} e'$), s'il existe une *trace* permettant depuis e, d'atteindre e'. Enfin, l'état e est atteignable sur le système de transitions, si e est atteignable à partir de l'état initial.

Nous désignons par State le type abstrait qui modélise les états et, par Transition celui qui modélise les transitions. Les notations suivantes sont utilisées : {T} dénote le type ensemble de T et, [T] dénote le type séquence de T. Ci-dessous, nous spécifions un jeu d'opérations sur ces types :

Operation : Transition ⟶ operation

{Operation (t) est l'opération associée à la transition t}

Dans la figure 3.3, Operation (<Attente de commande, ?(Ligne de commande), Attente de commande >) = ?(Ligne de commande).

Source : Transition ⟶ State

{Source(t) est l'état source de la transition t}

Dans la figure 3.3, Source (<Panier validé, !(Devis de commande), Devis envoyé>) = Panier validé.

Target : Transition ⟶ State

{Target(t) est l'état cible de la transition t}

Dans la figure 3.3, Target (<Panier validé, !(Devis de commande), Devis envoyé>) = Devis envoyé.

Inbounds : State ⟶ {Transition}

{Inbounds(e) est l'ensemble des transitions entrantes dans l'état e}

Dans la figure 3.3, Inbounds (Devis envoyé) = {<Panier validé, !(Devis de commande), Devis envoyé>} .

Outbounds : State ⟶ {Transition}

{Outbounds(e) est l'ensemble des transitions sortantes de l'état e}

Dans la figure 3.3, Outbounds (Devis envoyé>) = {<Devis envoyé, ?(Annulation de commande), e_f>, <Devis envoyé, ?(Réservation de commande), Commande réservée>} .

Nous définissons ci-dessous le type Trace et l'opération Traces sur celui-ci :

Type Trace : [Transition]

{Trace est le type qui modélise une séquence de transitions reliant deux états}

Traces: State x State ⟶ {Trace}

{Traces (e, e') est l'ensemble des traces permettant d'atteindre e' à partir de e}

Dans la figure 3.3, Traces (Devis envoyé, e_f) = {[<Devis envoyé, ?(Annulation de commande), e_f>], [<Devis envoyé, ?(Réservation de commande), Commande réservée>, Commande réservée, ?(Paiement), Paiement achevé>, <Paiement achévé, !(Bilan final), e_f >]}

3.3 Propriétés

Maintenant que nous avons décrit la modélisation de l'interface structurelle et comportementale des services à l'aide d'automates, nous allons définir des propriétés cette interface. Ces propriétés sont utilisées dans la suite de cette thèse dans les algorithmes de détection des incompatibilités entre deux interfaces. Des propriétés sur les transitions sont données dans la section 3.3.1. Des propriétés sur les états sont données par la section 3.3.2. La section 3.3.3 introduit la propriété d'interfaces équivalentes et celle d'interfaces réciproques, alors que la section 3.3.4 présente la propriété de services équivalentes et, celle de services compatibles.

3.3.1 Propriétés sur les transitions

Ci-dessous, nous donnons les propriétés d'équivalence et de réciprocité entre les transitions.

Définition 3.3.1 (Transitions équivalentes)

Soient t et t' deux transitions (t, t' : Transition), t et t' sont équivalentes, ce qui est noté t ≈ t', si et seulement si :

1. Polarity (Operation (t)) = Polarity (Operation (t'))
2. Name (Message (Operation (t))) = Name (Message (Operation (t')))
3. Type (Message (Operation(t))) ≡ Type (Message (Operation (t')))
4. Source (t) ≈ Source (t')

D'après la définition 3.3.1, deux transitions t et t' sont équivalentes si et seulement :

- l’opération associée à t et celle associée à t’ sont de même polarité (voir la ligne 1). C'est-à-dire, les deux opérations sont, soit des opérations d’envoi, soit des opérations de réception,

- le message envoyé ou reçu par l’opération associée à t et celui envoyé ou reçu par l’opération associée à t’ sont de même nom (voir ligne 2),

- le message envoyé ou reçu par l’opération associée à t et celui envoyé ou reçu par l’opération associé à t’ sont de types équivalents (voir ligne 3, voir également la définition de l’opérateur d’équivalence entre deux types donnée dans la section 3.2.1),

- enfin, l’état source de la transition t est équivalent à l’état source de la transition t’ (voir la définition 3.3.3 pour l’équivalence entre deux états).

Dans l’exemple de la figure 3.4, soit les deux messages Item et Product définis comme suit (voir la définition de type abstrait message fournie dans la section 3.2.1) :

- Item = <Order, tns : Item>, tel que Order représente le nom du message Item et tns : Item représente son type structurel,

- Product= <Order, tns : Product>, tel que Order représente le nom du message Product et tns : Product représente son type structurel.

Nous supposons que les types structurels tns : Item et tns : Product sont ceux données par le tableau 3.2 dans la section 3.2.1. Étant donné que l’algorithme de Zhang a donné un degré de similarité de 0.85 (supérieur à 0.8, seuil d’équivalence que nous avons fixé) entre ceux-ci (voir tableau 3.2.2), les deux types sont considérés équivalents (tns : Item ≈ tns : Product).

Ainsi, les deux transitions étiquetées respectivement par ?(Item) et ?(Product) dans les interfaces des fournisseurs 1 et 2, sont équivalentes car :

- L’opération associée à la transition étiquetée par ?(Item) et celle associée à la transition étiquetée par ?(Product) sont de même polarité (?),

- le message reçu par l’opération associée à la transition étiquetée par ?(Item), et celui reçu par l’opération associée à la transition étiquetée par ?(Product) sont de même nom (Order=Name(Message(?item) et Order=Name(Message(?Product))),

- le message reçu par l’opération associée à la transition étiquetée par ?(Item), et celui reçu par l’opération associée à la transition étiquetée par ?(Product) sont de types équivalents

(tns : Item ≡ tns : Product),

- enfin, les états sources e_0 et e'_0 de ces deux transitions sont équivalents (comme nous le verrons dans la définition 3.3.3, les états initiaux des automates de deux interfaces sont considérés comme équivalents).

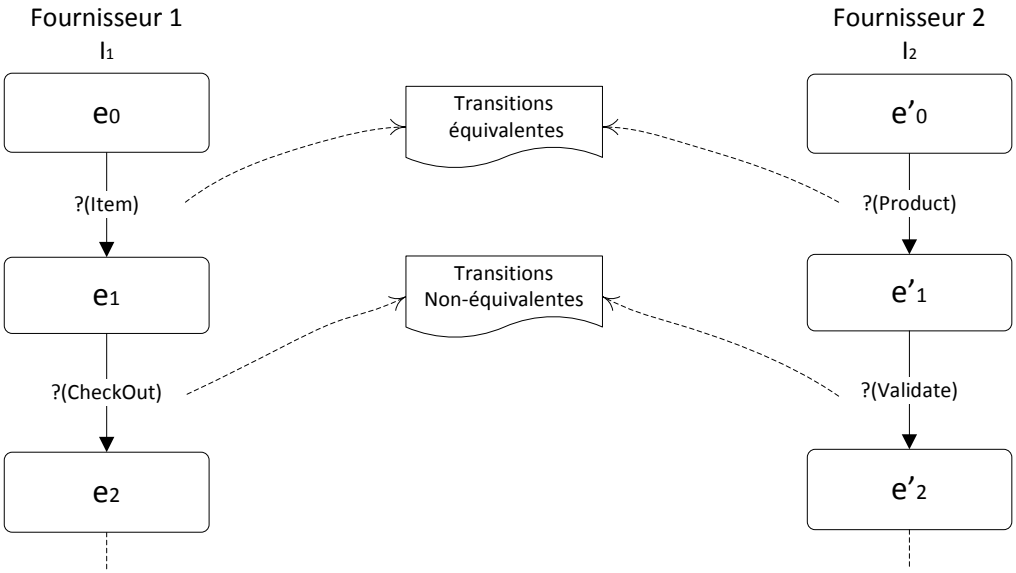

Figure 3.4 Equivalence entre transitions

Dans l'exemple de la figure 3.4, soit les deux messages CheckOut et Validate définis comme suit:

- CheckOut = <Check, tns : CheckOut>, tel que Check représente le nom du message CheckOut et tns : CheckOut représente son type structurel,

- Validate = <Validation, tns : Validate>, tel que Validation représente le nom du message Validate et tns : Validate représente son type structurel.

Dans la figure 3.4, les deux transitions étiquetées respectivement par ?(CheckOut) et ?(Validate) dans les interfaces des fournisseurs 1 et 2, ne sont équivalentes car le nom (Check) du message reçu par l'opération associée à la transition étiquetée par ?(CheckOut) est différent du nom (Validation) du message reçu par l'opération associée à la transition étiquetée par ?(Validate).

Définition 3.3.2 (Transitions réciproques)

Soient t et t' deux transitions (t, t' : Transition), t et t' sont réciproques, ce qui est noté t ≋

t’, si et seulement si :

1. Polarity (Operation (t)) = $\overline{\text{Polarity (Operation (t'))}}$
2. Name (Message (Operation (t))) = Name (Message (Operation (t’)))
3. Type (Message (Operation(t))) ≡ Type (Message (Operation (t’)))
4. Source (t) ≋ Source (t’)

D’après la définition 3.3.2, deux transitions t et t’ sont réciproques si et seulement les conditions suivantes sont vérifiées :

- l’opération associée à t et celle associée à t’ sont de polarité réciproques (voir ligne 1),
- le message envoyé ou reçu par l’opération associée à t et celui envoyé ou reçu par l’opération associée à t’ sont de même nom (voir ligne 2),
- le message envoyé ou reçu par l’opération associée à t et celui envoyé ou reçu par l’opération associé à t’ sont de types équivalents (voir ligne 3),
- enfin, les deux états sources respectivement de t et t’ sont réciproques l’un de l’autre (voir la définition 3.3.4 pour la réciprocité entre deux états).

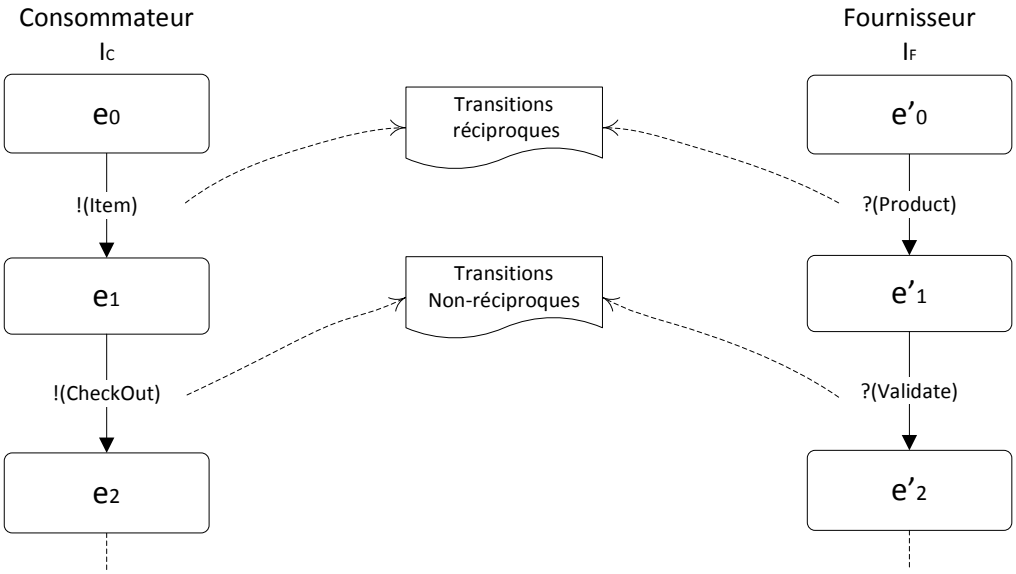

Figure 3.5 Réciprocité entre transitions

Dans l’exemple de la figure 3.5, soit les deux messages Item et Product sont les mêmes

de l'exemple de la figure 3.4. Ainsi, les deux transitions étiquetées respectivement par !(Item) et ?(Product) respectivement dans les interfaces du consommateur et du fournisseur, sont réciproques car :

- La polarité ! de l'opération associée à la transition étiquetée par !(Item) est la réciproque de la polarité ? de l'opération associée à la transition ?(Product).

- le message envoyé par l'opération associée à la transition étiquetée par !(Item), et celui reçu par l'opération associée à la transition étiquetée par ?(Product) sont de même nom (Order=Name(Message(!item) et Order=Name(Message(?Product))),

- le message envoyé par l'opération associée à la transition étiquetée par !(Item), et celui reçu par l'opération associée à la transition étiquetée par ?(Product) sont de types équivalents (tns : Item ≡ tns : Product),

- enfin, les états sources e_0 et e'_0 de ces deux transitions sont réciproques (comme nous le verrons dans la définition 3.3.4, les états initiaux des automates de deux interfaces sont considérés comme réciproques).

Cependant, les deux transitions étiquetées respectivement par !(CheckOut) et ?(Validate) dans les interfaces du consommateur et du fournisseur, ne sont réciproques car le nom (Check) du message envoyé par l'opération associée à la première transition est différent du nom (Validation) du message reçu par l'opération associée à la deuxième transition.

3.3.2 Propriétés sur les états

Nous donnons maintenant les propriétés d'équivalence et de réciprocité entre les états.

Définition 3.3.3 (Etats équivalents)

Soient e et e' deux états (e, e' : State), e et e' sont équivalents, ce qui est noté e ≈ e', si et seulement si ∀ t ∈ Inbounds(e) ∃ t' ∈ Inbounds(e') tel que t ≈ t'.

D'après cette définition, deux états sont équivalents si chaque transition entrante dans l'un est équivalente à une transition entrante dans l'autre.

Cas particuliers : les états sources (respectivement finaux) des automates de deux interfaces sont considérés équivalents deux à deux.

Dans la figure 3.4, les deux états e_1 et e'_1 sont équivalents car les deux transitions

entrantes étiquetées ?(Item) et ?(Product) entrantes respectivement dans e_1 *et* e'_1 sont équivalentes (voir la démonstration de ceci dans l'exemple de la définition 3.3.1). Cependant, les états e_2 et e'_2 ne sont pas équivalents car les deux transitions étiquetées par ?(CheckOut) et ?(Validate), entrantes respectivement dans les deux états e_2 et e'_2, ne sont pas équivalentes (voir la démonstration de ceci dans l'exemple exemple de la définition 3.3.1).

Définition 3.4.4 (Etats réciproques)

Soient e et e' deux états (e, e' : State), e et e' sont réciproques, ce qui est noté e $\approx$ e', si et seulement si $\forall$ t $\in$ Inbounds(e) $\exists$ t' $\in$ Inbounds(e') tel que t $\approx$ t'.

D'après cette définition, deux états sont réciproques si chaque transition entrante dans l'un est réciproque à une transition entrante dans l'autre.

Cas Particuliers : les états sources (respectivement finaux) des automates de deux interfaces sont considérés réciproques deux à deux.

Dans la figure 3.5, les deux états e_1 et e'_1 sont réciproques car les deux transitions entrantes étiquetées !(Item) et ?(Product), entrantes respectivement dans e_1 *et* e'_1 sont réciproques (voir la démonstration de ceci dans l'exemple de la définition 3.3.2). Cependant, les états e_2 et e'_2 ne sont pas réciproques car les deux transitions étiquetées par !(CheckOut) et ?(Validate), entrantes respectivement dans les deux états e_2 et e'_2, ne sont pas équivalentes (voir la démonstration de ceci dans l'exemple exemple de la définition 3.3.2).

3.3.3 Propriétés sur les interfaces

Dans cette section, nous donnons les propriétés d'équivalence et de réciprocité entre les interfaces.

Interfaces équivalentes

Deux interfaces décrivant deux services sont équivalentes si et seulement si toute trace acceptée (voir la définition de trace acceptée dans la section 3.2.2) par l'automate de l'un est aussi acceptée par l'automate de l'autre, et réciproquement. Dans la figure 3.4, la trace [<e_0, ?(Item), e_1>, < e_1, ?(CheckOut), e_2>] est acceptée par l'interface du Fournisseur 1 mais ne l'est pas par l'interface du Fournisseur 2.

Nous désignons par Interface le type abstrait de l'interface de service donnée par la définition 3.2.1. La notion suivante est utilisée, I.Transistion représente l'ensemble des

transitions d'une interface I.

Définition 3.3.5 (Interfaces équivalentes)

Soient I et I' deux interfaces (I, I' : Interfaces), I et I' sont équivalentes si et seulement si ∀ t ∈ I.Transition ∃ t' ∈ I'.Transition telle que t ≈ t'.

D'après cette définition, des deux interfaces I et I' sont équivalentes si pour toute transition t de I il existe une transition t' de I' équivalente à t (t ≈ t').

Interfaces réciproques

D'un point de vue informel, deux interfaces I et I' sont réciproques si quelque soit l'état dans lequel se trouvent les deux services modélisés par I et I', chaque message envoyé par l'un doit pouvoir reçu par l'autre, et *vice-versa*.

Définition 3.4.6 (Interfaces réciproques)

Soient I et I' deux interfaces (I, I' : Interfaces), I et I' sont réciproques si et seulement si ∀ t ∈ I.Transition ∃ t' ∈ I'.Transition telle que t ≋ t'.

D'après cette définition, des deux interfaces I et I' sont réciproques si pour toute transition t de I il existe une transition t' de I' réciproque à t (t ≋ t').

3.3.4 Propriétés sur les services

Dans cette section, nous donnons les propriétés de substitution et de compatibilité entre deux services.

Définition 3.4.7 (Services substituables)

Deux services dont les interfaces sont équivalentes sont substituables : tout consommateur du premier peut interagir correctement avec le deuxième sans recourir à une adaptation.

Définition 3.4.8 (Services compatibles)

Deux services dont les interfaces sont réciproques sont compatibles : l'un peut interagir correctement avec l'autre.

3.4 Conclusion

Dans ce chapitre, nous avons présenté notre proposition pour la modélisation de l'interface structurelle et comportementale des services à l'aide des systèmes de transitions étiquetées.

Nous avons définis sur la base de cette modélisation, des propriétés sur les interfaces et les services, notamment les propriétés d'interfaces équivalentes, d'interfaces réciproques, de services substituables, et de services compatibles. Ces propriétés sont utilisées dans les chapitres suivants de ce manuscrit.

Une implantation de la modélisation proposée est donnée dans le chapitre 5.

Dans l'état de l'art, différentes approches ont proposé de modéliser les interfaces de service à l'aide des systèmes de transition étiquetés afin d'étudier leur équivalence et leur compatibilité [BCT+06, BSBM04, CLB08]. Cependant, ces approches ne prennent pas en considération les aspects structurels de messages. Seuls les aspects comportementaux de ceux-ci sont considérés. Dans notre définition pour la compatibilité et l'équivalence entre des interfaces, nous avons considérés aussi bien les aspects comportementaux que les aspects structurels de messages.

Dans le chapitre suivant, nous proposons une méthode pour la détection des incompatibilités entre deux interfaces et, la génération automatique des adaptateurs.

CHAPITRE 4

Adaptation d’interactions de services par détection et résolution des incompatibilités

Sommaire

4.1 Introduction

Dans le chapitre précédent, nous avons proposé une modélisation de l’interface structurelle et comportementale des services Web à l’aide d’automates. En nous appuyant sur cette modélisation nous avons proposé des propriétés telles que la réciprocité et l’équivalence d’interfaces afin de vérifier la compatibilité et la substituabilité entre les services.

Dans ce chapitre, nous proposons une méthode pour la détection et la résolution des incompatibilités. Il s’agit non seulement de déterminer si deux services ne sont pas compatibles, mais le cas échéant de détecter leurs incompatibilités. Ensuite, de générer un adaptateur pour la résolution des incompatibilités détectées.

La section 4.2 décrit les principes généraux de notre proposition à travers un exemple illustratif. La section 4.3 détaille notre solution pour la détection des incompatibilités entre

deux interfaces. Dans la section 4.4, nous discutons de notre méthode à bases des patrons pour la résolution des incompatibilités par génération automatique des adaptateurs. Enfin, la section 4.5 conclut le chapitre.

4.2 Principes généraux de la proposition

Dans cette section nous présentons les principes généraux de notre proposition à travers une illustration. La section 4.2.1 présente un exemple illustratif, alors que les sections 4.2.2 et 4.2.3 illustrent respectivement les principes de détection et de résolution des incompatibilités entre deux services.

4.2.1 Illustration

La figure 4.1 illustre un scénario d'achat des produits en ligne. Le scénario inclut deux participants : le service Consommateur et, le service Fournisseur 1. Dans cet exemple, nous considérons que le service Consommateur a été développé spécifiquement pour interagir avec le service Fournisseur 1. Cela veut dire que l'interface *requise* par le service Consommateur a été conçue de manière qu'elle soit *réciproque* à celle *fournie* par le service Fournisseur 1.

La figure 4.1 présente les interfaces modélisées en automates des deux services Consommateur et Fournisseur 1. Pour placer une commande de n articles, le service Consommateur exécute n fois l'opération !(Ligne de commande). Chaque exécution se traduit par un envoi d'un message contenant les détails d'un et d'un seul article de la commande. L'arrivée du message envoyé par le consommateur au service fournisseur se traduit par l'exécution de l'opération ?(Ligne de commande) qui permet de consommer le message reçu. Une fois toute les lignes de commande envoyées, il s'agit d'exécuter l'opération !(Valider panier) qui se traduit par un envoi d'un message signalant la terminaison de la commande auprès du service fournisseur. Une fois le message arrivé, l'opération ?(Valider panier) de l'interface de service fournisseur est exécutée, ce qui permet au fournisseur de procéder au calcul du devis de la commande et de le retourner au consommateur en exécutant l'opération !(Devis). A cette opération correspond l'opération ?(Devis) du coté du service Consommateur. Une fois le devis reçu, le consommateur peut soit, annuler la commande par l'exécution de l'opération !(Annuler) soit, régler la commande par l'exécution de l'opération !(Paiment-Adresse) qui se traduit par l'envoi des détails de la carte bancaire du client au service fournisseur, ainsi que l'adresse à laquelle il souhaite recevoir la facture.

Supposons qu'à un moment donné, le service Fournisseur 1 ne soit plus disponible.

L’administrateur du service Consommateur décide alors de substituer le service Fournisseur 1 par le service Fournisseur 2, qui répond aux mêmes besoins que Fournisseur 1.

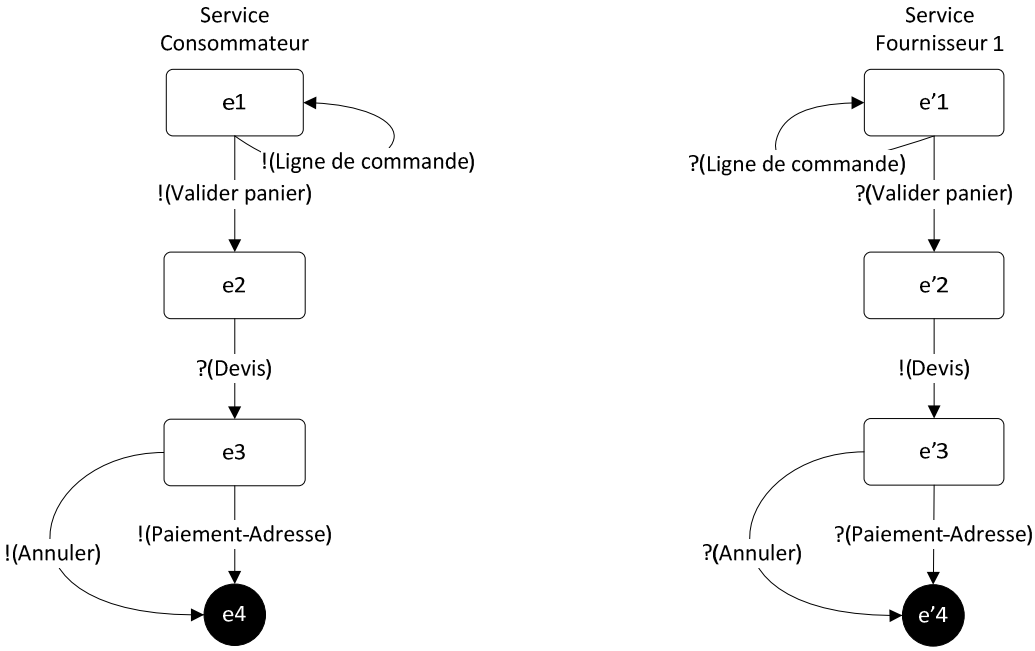

Figure 4.1 deux automates décrivant deux interfaces compatibles

Supposons qu’à un moment donné, le service Fournisseur 1 ne soit plus disponible. L’administrateur du service Consommateur décide alors de substituer le service Fournisseur 1 par le service Fournisseur 2, qui répond aux mêmes besoins que Fournisseur 1.

Étant donné que le service Consommateur a été spécifié pour interagir avec le service Fournisseur 1, des incompatibilités sont susceptibles de se manifester entre l’interface du service Consommateur et celle du service Fournisseur 2. Ce qui rend les interactions impossibles entre les deux services.

Pour remédier à ce problème, notre solution propose la mise en œuvre d’un adaptateur d’interactions généré automatiquement. Cette génération se base sur les principes de détection et de résolution des incompatibilités.

4.2.2 Principe de détection des incompatibilités

Il s’agit de comparer les deux interfaces modélisées en automates des deux services Consommateur et Fournisseur 2 pour identifier les incompatibilités mixtes qui font que ceux-ci sont incompatibles.

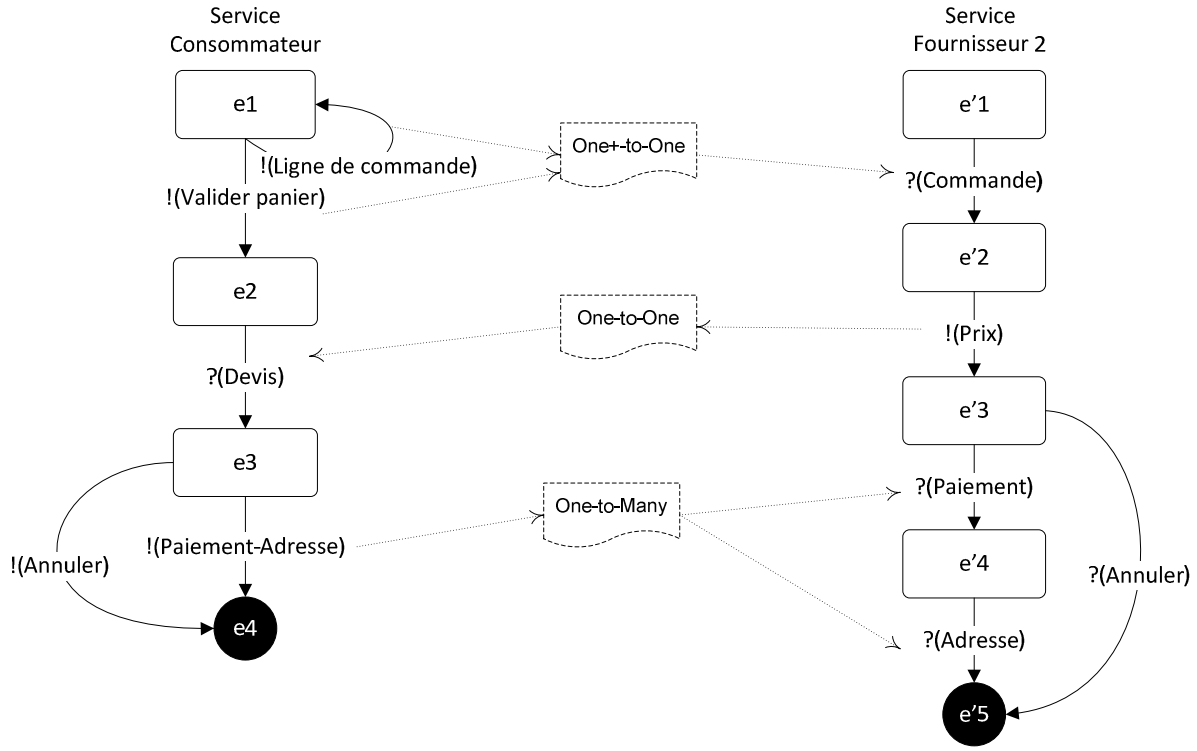

Figure 4.2 Détection des incompatibilités entre deux interfaces incompatibles

La figure 4.2 illustre la détection des incompatibilités, citées ci-après, entre l'interface de services Consommateur et Fournisseur 2 :

- Alors que le service Consommateur prévoit la réception du message Devis = <Cotation, tns :Devis> ayant comme Cotation et comme structure tns :Devis, le service Fournisseur 2 envoie le message Prix = <Cotation, tns :Prix> dont la structure tns :Prix n'est pas équivalente à celle du message Devis (En supposant dans cet exemple que deux structures de noms différents signifie qu'elles ne sont pas équivalentes). Ce type d'incompatibilité est caractérisé par le patron One-to-One (voir la section 4.3.1) : à une opération du consommateur correspond une opération du fournisseur, mais le message attendu n'a pas une structure équivalente à celle du message envoyé.

- Pour traiter une commande de n articles, le service Fournisseur 2 prévoit la réception d'un seul message Commande dont la structure regroupe toutes les lignes de la commande. Cependant, le consommateur ne peut envoyer qu'une seule ligne de commande par message !(Ligne de commande). Ce type d'incompatibilité est caractérisé par le patron One^{+}-to-One car l'exécution n fois de l'opération !(Ligne de commande) de l'interface du service Consommateur correspond à une exécution unique de l'opération ?(commande) de l'interface du service Fournisseur 2.

- Alors que le service Consommateur permet le paiement de la commande et l'envoi de l'adresse de facturation par l'exécution d'une seule opération !(PaiementAdresse), le service Fournisseur 2 prévoit la réception des deux messages différents : l'un pour le paiement ?(Paiement) et l'autre pour l'adresse de facturation ?(Adresse). Nous caractérisons ce type d'incompatibilité par le patron One-to-Many car l'exécution d'une opération de l'interface du premier service correspond à l'exécution de deux ou plusieurs opérations différentes deux à deux de l'interface du seconde service.

La liste de ces patrons est complétée et discutée plus loin dans la section 4.3.1.

4.2.3 Principe de résolution des incompatibilités

Le principe de la résolution des incompatibilités entre deux services consiste à mettre en place un adaptateur des interactions. La figure 4.3 illustre son principe : il intercepte et adapte les messages échangés entre les deux services Consommateur et Fournisseur 2. Cela permet de préserver ces services intacts ; il n'agit que sur les messages échangés entre ceux-ci.

Un adaptateur d'interactions entre deux services est perçu comme une boite noire qui consomme des messages en termes de l'interface du service émetteur, et transmet des messages en termes de l'interface du service destinataire. La transformation des messages échangés se fait au sein de l'adaptateur de manière transparente pour les services partenaires.

Pour résoudre l'incompatibilité du type One^{+}-to-One, l'adaptateur exécute la séquence des opérations suivantes :

[(?(Ligne de commande))^{n} , ?(valider panier) , (commande = $F_{aggregation}$(Ligne de commande$_1$, …, Ligne de commande$_n$)) , !(Commande))]

La consommation d'un message concernant une ligne de commande et provenant du service Consommateur, se traduit par le déclenchement de la transition ?(Ligne de commande) de l'adaptateur (voir la figure 4.3). Une fois cette transition déclenchée, l'adaptateur reste toujours à l'état a_0, ce qui lui permet de consommer un autre message concernant une ligne de commande ou bien, un message de validation de commande. L'indice n représente le nombre de messages consommés. Une fois qu'un message concernant la validation de panier est consommé ?(Valider panier), l'adaptateur passe à l'état a_1. A ce stade, l'agrégation des messages consommés en un seul est effectuée par l'application de la fonction de transformation $F_{aggregation}$ sur les n messages reçus. Puis, l'adaptateur passe à l'état a_2. A cet état, l'adaptateur envoie le message Commande issu de la fonction d'agrégation, et cela par le

déclenchement de la transition !(Commande) (voir la figure 4.3).

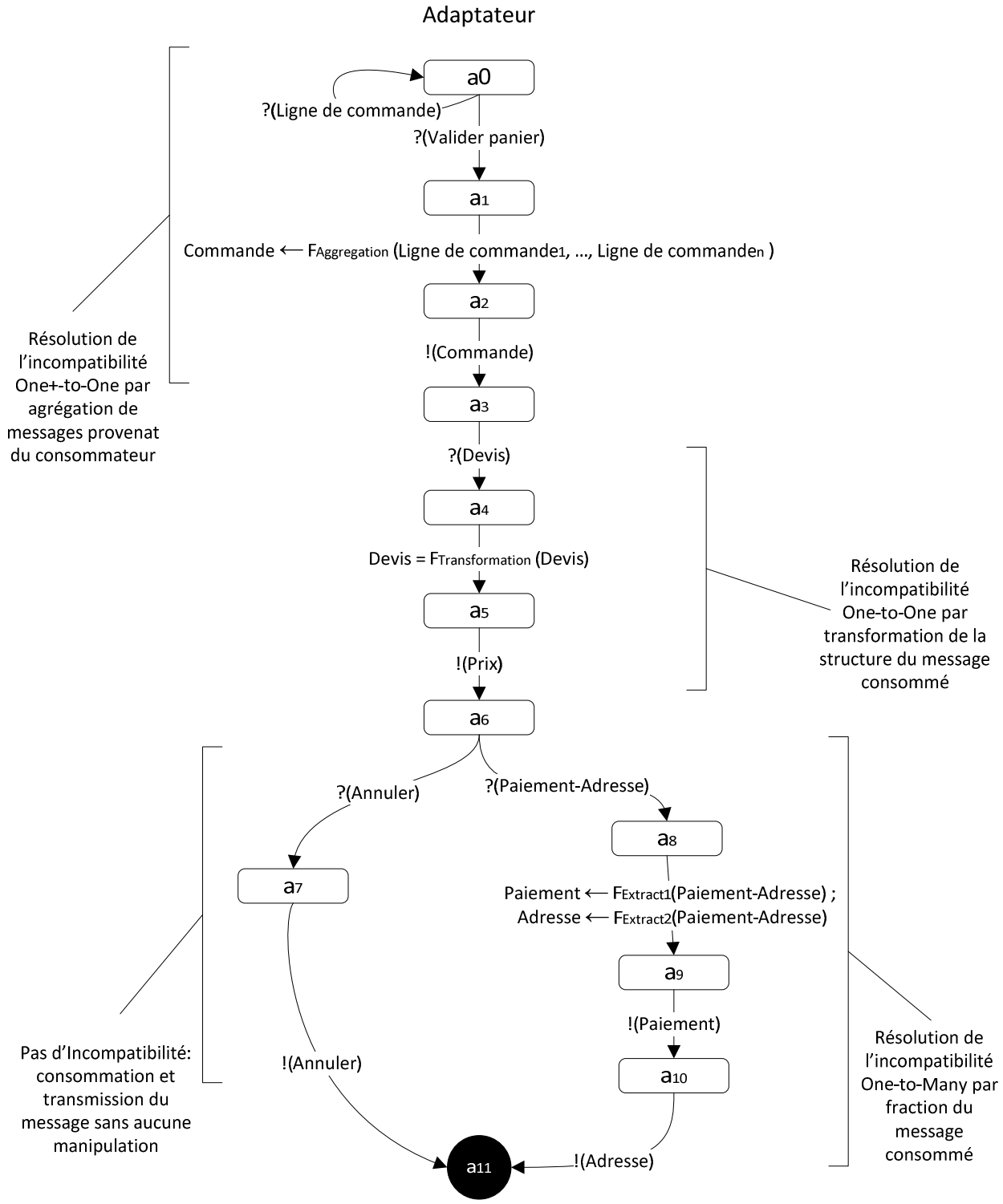

Figure 4.3 Automate de l'adaptateur des interactions de services Consommateur et Fournisseur 2

4.3 Détection des incompatibilités

Nous proposons de formaliser la détection des incompatibilités par la fourniture de patrons, chacun caractérisant un type d'incompatibilités. La reconnaissance des patrons d'incompatibilités s'appuie sur un algorithme qui parcourt de manière synchrone les états et les transitions de deux automates modélisant les interfaces soient I et I', et retourne l'ensemble de leurs incompatibilités [AB08]. Au niveau de chaque paire d'états soit (e,e') tel que e dans I et e' dans I', l'algorithme vérifie si une incompatibilité est détectée entre les transitions sortantes de e et celles de e'.

Nous désignons par Pattern, et Incompatibility les types abstraits qui modélisent respectivement les patrons, et les incompatibilités. Ces types sont donnés ci-dessous :

Type enum Pattern : { One-to-One, One-to-Many, Many-to-One, One^{+}-to-One, One-to-One^{+}}
{Pattern est le type énuméré qui consiste en des chaines de caractères représentant les noms de patrons d'incompatibilités}

type Incompatibility ⟨

T : [Transition]	{Liste des transitions dont les opérations associées sont de polarité !}
T' : [Transition]	{Liste des transitions dont les opérations associées sont de polarité ?}
P : Patttern	{Le patron qui relie T à T'}

⟩

Nous spécifions l'opérateur Detection comme suit :

Detection : Interface x Interface ⟶ [Incompatibility]
{Detection(I,I') est la liste des incompatibilités détectées entre les deux interfaces I et I'}

Dans l'exemple de la figure 4.2 (dans la section 4.2.2), l'opérateur Detection appliqué sur l'interface du service Consommateur et celle du service Fournisseur 2 donne la liste Inc des trois incompatibilités illustrées sur la figure telles que (Inc[i] représente les i^{ème} élément de la liste) :

Inc [0] = <[!(ligne de commande), !(Valider Panier)], [?(Commande)], One^{+}-to-One>

Inc [1] = <[!(Prix)], [?(Devis)], One-to-One>

Inc [2] = <[!(PaiementAdresse)], [?(Paiment), ?(Adresse)], One-to-Many>

Dans la section 4.3.1, nous caractérisons les incompatibilités par des expressions logiques sur les automates. Dans la section 4.3.2, nous étudions la complétude de chaque expression par rapport au patron de l'incompatibilité qu'elle modélise, ainsi leur exclusion mutuelle. La section 4.3.3 présente l'algorithme de détection des incompatibilités.

4.3.1 Patrons des incompatibilités

Nous caractérisons des incompatibilités entre deux interfaces modélisées par d'automates par des expressions logiques sur les automates. L'algorithme de détection des incompatibilités évalue chacune de des expressions sur le couple d'états courants lors de la comparaison des deux automates décrivant les deux interfaces. Une fois une incompatibilité détectée, l'algorithme détermine le nouveau couple d'états courants et continue son parcourt. La fonction d'évaluation des expressions logiques EvaluateLogicExpressions et celle de détermination du nouveau couple d'états courants Suivant sont spécifiées comme suit :

EvaluateLogicExpressions : State x state ⟶ Incompatibility

{EvaluateLogicExpressions(e,e') retourne l'incompatibilité i = <T, T', P> détectée entre les listes de transitions T et T' au niveau du couple d'états (e,e')}

Suivant : Incompatibility ⟶ State x state

{Suivant(I) retourne le nouveau couple d'états courants (e, e') à partir de lequel l'algorithme continue son parcourt}

Nous exprimons ici l'ensemble des patrons des expressions logiques que nous proposons :

Patron d'incompatibilité de type One-to-One :

D'après la figure 4.2, une incompatibilité de type One-to-One est détectée entre les transitions !(Prix) et ?(Devis). Cette incompatibilité est engendrée du fait que la structure tns : Prix du message Prix attendu par le service Consommateur n'est pas équivalente à la structure tns : Devis du message Devis envoyé par le service Fournisseur 2.

En comparant les transitions sortantes des états courants e et e' respectivement dans I et I', une incompatibilité de type One-to-One est détectée, si l'expression booléenne suivante est évaluée à vrai :

1. $\exists\, t \in \text{Outbounds}(e) \wedge \exists\, t' \in \text{Outbounds}(e') :$
2. $\text{Polarity}(\text{Operation}(t)) = \overline{\text{Polarity}(\text{Operation}(t'))} \wedge$
3. $\text{Name}(\text{Message}(\text{Operation}(t))) = \text{Name}(\text{Message}(\text{Operation}(t'))) \wedge$
4. $\text{Type}(\text{Message}(\text{Operation}(t))) \not\equiv \text{Type}(\text{Message}(\text{Operation}(t')))$

D’après l’expression ci-dessus, une incompatibilité de type One-to-One est détectée entre deux transitions t et t’ qui sont sortantes respectivement de e et e’ (voir ligne 1), si et seulement si :

- le message envoyé ou reçu par l’opération associée à t, et celui envoyé ou reçu par l’opération associé à t’ sont de polarités réciproques (voir ligne 2),

- le message envoyé ou reçu par l’opération associée à t, et celui envoyé ou reçu par l’opération associé à t’ sont de même nom (voir ligne 3),

- le type structurel du message envoyé ou reçu par l’opération associée à t, et celui du message envoyé ou reçu par l’opération associé à t’ ne sont pas équivalents (c'est-à-dire le degré de similarité donné par l’algorithme de Zhang [ZYGW06] est inférieur ou égale à 0.8) (voir ligne 4).

Dans le cas où l’expression logique du patron One-to-One (présentée ci-dessus) est évaluée à vrai pour les transitions t et t’ sortantes respectivement de e et e’ (soit (e, e’) le couple d’états courants), la fonction EvaluateLogicExpressions retourne l’incompatibilité i=<T, T’, P> sous la forme suivante :

- T comprend un seul élément soit T[0], tel que T[0] = t

- T’ comprend un seul élément soit T’[0], tel que T’[0]= t’

- P = One-to-One

Ainsi, le nouveau couple d’états courants est déterminé comme suit : Suivant(i) = (Target(i.T[0]), Target(i.T’[0])). C'est-à-dire le nouveau couple d’états courants sont les états destinataires de T[0] et T’[0].

Patron d’incompatibilité de type One-to-Many:

D’après la figure 4.2, une incompatibilité de type One-to-Many est détectée entre la transition !(PaiementAdresse) d’une part et les transitions ?(Paiment) et ?(Adresse) d’une autre part. Cette incompatibilité est engendrée par le fait que le message d’envoi comporte des informations à propos du paiement de la commande ainsi que l’adresse de facturation. Cependant le service destinataire prévoit que ces informations arrivent dans deux messages différents.

Ci-dessous, l’expression booléenne qui modélise cette incompatibilité entre I et I’ aux

états e ∈ I et e’ ∈ I’ :

1. $\exists$ t $\in$ Outbounds(e) $\wedge$ $\exists$ tr $= [t'_1, t'_2, \dots, t'_n] \in$ Traces (e', e'_n) :
2. Polarity (Operation(t)) $= !$ $\wedge$ $(\bigwedge_{i=1}^{n}$ Polarity(Operation$(t'_i)) = ?) \wedge$
3. $\neg(\bigvee_{i=1}^{n}$ Type(Message(Operation(t))) $\equiv$ Type (Operation$(t'_i))) \wedge$
4. Type(Message(Operation(t))) $\equiv$ $\bigcup_{i=1}^{n}$(Type (Operation(t'_i)))

D’après l’expression ci-dessus, une incompatibilité de type One-to-Many est détectée entre une transition t de I et une trace des transitions $tr = [t'_1, t'_2, \dots, t'_n]$ de I’ (voir ligne 1), si et seulement si :

- l’opération associée à t est de polarité d’envoi (voir ligne 2, Polarity(t) = !), alors que les opérations associées aux transitions de la trace tr sont de polarité de réception. (voir ligne 2, $\bigwedge_{i=1}^{n}$ Polarity$(t'_i) = ?$),

- aucun message associé à une transition de la trace tr n’a un type équivalent à celui du message associé à la transition t (voir ligne3),

- le type structurel du message envoyé par l’opération associée à la transition t est équivalent à celui obtenu par l’union (définis par Zhang et Coll. [LSZ+06]) de types de messages attendus par les opérations associées associés aux transitions de la trace tr (voir ligne 4).

Dans le cas où l’expression logique du patron One-to-Many (présentée ci-dessus) est évaluée à vrai pour la transition t sortante de e et la trace $tr = [t'_1, t'_2, \dots, t'_n]$ dont t'_1 est sortante de e’ (soit (e, e’) le couple d’états courants), la fonction EvaluateLogicExpressions retourne l’incompatibilité i = <T, T’, P> sous la forme suivante :

- T comprend un seul élément soit T[0], tel que T[0] = t

- T’ comprend la trace tr, tel que T’[i]= t'_i

- P = One-to-Many

Ainsi, le nouveau couple d’états courants suite est déterminé comme suit : Suivant(i) = (Target(i.T[0]), Target(i.T’[n])). C'est-à-dire le nouveau couple d’états courants sont les états destinataires de T[0] et de la dernière transition T’[n] de la liste T’.

Patron d’incompatibilité de type Many-to-One :

Une incompatibilité de type Many-to-One se manifeste lorsque n opérations d’envoi de

message d'un service d'interface I correspondent à une action de réception de message d'un autre service d'interface I'.

Ci-dessous l'expression booléenne qui modélise cette incompatibilité entre I et I' aux états e ∈ I et e' ∈ I' :

1. $\exists\ tr = [t_1, \dots, t_n] \in \text{Traces}\ (e, e_n) \wedge \exists\ t' \in \text{Outbounds}(e')$:
2. $(\bigwedge_{i=1}^{n} \text{Polarity}(\text{Operation}(t_i)) = !) \wedge \text{Polarity}\ (\text{Operation}(t')) = ? \wedge$
3. $\neg(\bigvee_{i=1}^{n} \text{Type}(\text{Message}(\text{Operation}(t_i))) \equiv \text{Type}(\text{Message}(\text{Operation}(t'))) \wedge$
4. $\bigcup_{i=1}^{n}(\text{Type}(\text{Message}(\text{Operation}(t_i))) \equiv \text{Type}(\text{Message}(\text{Operation}(t'))))$

D'après cette expression, une incompatibilité de type Many-to-One est détectée entre une trace tr sortante de l'état courant e de I et une transition t' sortante de l'état courant e' de I' (voir ligne 1), si et seulement si :

- les opérations associées aux transitions de la trace tr sont des polarité d'envoi (voir ligne 2, $\bigwedge_{i=1}^{n} \text{Polarity}(t_i) = !$), alors que l'opération associée à la transition t' est de polarité de réception (voir ligne 2, Polarity (t') = ?),

- aucun message attendu par l'opération associée à une transition de la trace tr n'a un type équivalent à celui du message envoyé par l'opération associée à la transition t (voir ligne 3),

- le type structurel obtenu par l'union de types de messages envoyés par les opérations associées aux transitions de la trace tr est équivalent à celui du message attendu par l'opération associée à la transition t' (voir ligne 4).

Dans le cas où l'expression logique du patron Many-to-One (présentée ci-dessus) est évaluée à vrai pour la trace $tr = [t_1, \dots, t_n]$ dont t_1 est sortante de e et la transistion t' sortante de e' (soit (e, e') le couple d'états courants), la fonction EvaluateLogicExpressions retourne l'incompatibilité i = <T, T', P> sous la forme suivante :

- T comprend la trace tr, tel que T[i]= t_i

- T' comprend un seul élément T'[0], tel que T'[0] = t'

- P = Many-to-One

Ainsi, le nouveau couple d'états courants est déterminé comme suit : Suivant(i) = (Target(i.T[n]), Target(i.T'[0])). C'est-à-dire nouveau couple d'états courants sont les états destinataires de T[n] et de T'[0].

Patron d'incompatibilité de type One^{+}-to-One:

D'après la figure 4.2, une incompatibilité de type One^{+}-to-One est détectée entre les transitions !(Ligne de commande) et !(Valider panier) d'une part et la transition ?(Commande) d'une autre part.

Ci-dessous l'expression booléenne qui modélise cette incompatibilité entre I et I' aux états e ∈ I et e' ∈ I' :

1. $\exists\, t_1, t_2 \in \text{Outbounds}(e) \wedge \exists\, t' \in \text{Outbounds}(e') :$
2. $\text{Target}(t_1) = \text{Source}(t_1) \wedge \text{Target}(t_2) \neq \text{Source}(t_2) \wedge$
3. $\text{Target}(t') \neq \text{Source}(t') \wedge$
4. $\text{Polarity}(\text{Operation}(t_1)) = ! \wedge \text{Polarity}(\text{Operation}(t_2)) = ! \wedge$
5. $\text{Polarity}(\text{Operation}(t')) = ? \wedge$
6. $\text{Type}(\text{Message}(\text{Operation}(t_1))) \equiv \text{Type}(\text{Message}(\text{Operation}(t'))) \wedge$
7. $\text{MaxOccurs}(\text{Message}(\text{Operation}(t))) = 1 \wedge$
8. $\text{MaxOccurs}(\text{Message}(\text{Operation}(t'))) = *$

D'après l'expression ci-avant, une incompatibilité de type One^{+}-to-One est détectée entre deux transitions t_1 et t_2 sortantes de l'état courant e de I et une transition t' sortante de l'état courant e' de I' (voir ligne 1), si et seulement si :

- l'état source de t_1 est l'état destinataire lui-même (voir ligne 2, $\text{Target}(t_1) = \text{Source}(t_1)$), et l'état cible de t_2 et celui de t' sont différents de leurs états sources (ligne 2 et 3, $\text{Target}(t_2) \neq \text{Source}(t_2) \wedge \text{Target}(t') \neq \text{Source}(t')$),

- les opérations associées respectivement à t_1 et t_2 sont de polarité ! (voir ligne 4), alors que l'opération associée à t' est de polarité ? (voir ligne 5).

- le type du message envoyé par l'opération associée à t_1 est équivalent à celui du message attendu par l'opération associée à t' (voir ligne 6).

- le message envoyé par l'opération associée à t_1 ne peut porter qu'un seul élément du type Type (Message(Operation(t))) (voir ligne 7) (voir ci-dessous la spécification de l'opération MaxOccurs), alors que le message attendu par l'opération associée à t' peut plusieurs messages du type Type(Message(Operation(t'))) (voir ligne 8).

MaxOccurs est l'opérateur spécifié comme suit :

MaxOccurs : message ⟶ Integer

{MaxOccurs(m) est le nombre des éléments du type Type(m) que le message m peut

comprendre. Ce nombre est donné dans la description XML du message }

Dans le cas où l'expression logique du patron One$^+$-to-One (présentée ci-dessus) est évaluée à vrai pour les transition t_1 et t_2 sortantes de e d'une part et la transition t' sortante de e' d'autre part (soit (e, e') le couple d'états courants), la fonction EvaluateLogicExpressions retourne l'incompatibilité i = <T, T', P> sous la forme suivante :

- T comprend les transistion t_1 et t_2, tel que T= [t_1, t_2]
- T' comprend un seul élément T'[0], tel que T'[0] = t'
- P = One$^+$-to-One

Ainsi, le nouveau couple d'états courants est déterminé comme suit : Suivant(i) = (Target(i.T[1]), Target(i.T'[0])). C'est-à-dire nouveau couple d'états courants sont les états destinataires de T[1] et de T'[0].

Patron d'incompatibilité de type One-to-One$^+$:

Une incompatibilité de type One-to-One$^+$ se manifeste lorsque le message envoyé par l'opération associée à une transition t de I peut contenir plusieurs élément de type Type (Message(Operation(t))), alors que le message reçu par l'opération associée à une transition t' de I' ne peut contenir qu'un seul message de Type(Message(t')) tel que Type(Message(Operation(t))) ≡ Type(Message(Operation(t'))).

Ci-dessous l'expression booléenne qui modélise cette incompatibilité entre I et I' aux états e ∈ I et e' ∈ I' :

1. $\exists\, t \in \text{Outbounds}(e) \wedge \exists\, t' \in \text{Outbounds}(e') :$
2. $\text{Target}(t) \neq \text{Source}(t) \wedge \text{Target}(t') = \text{Source}(t') \wedge$
3. $\text{Polarity}\big(\text{Operation}(t)\big) = ! \wedge \text{Polarity}(\text{Operation}(t')) = ? \wedge$
4. $\text{Type}(\text{Message}(\text{Operation}(t))) \equiv \text{Type}(\text{Message}(\text{Operation}(t'))) \wedge$
5. $\text{MaxOccurs}(\text{Message}(\text{Operation}(t))) = * \wedge$
6. $\text{MaxOccurs}(\text{Message}(\text{Operation}(t'))) = 1$

D'après cette expression, une incompatibilité de type One-to-One$^+$ est détectée entre une transition t sortante de l'état e de I et une transition t' sortante de l'état e' de I' (voir ligne 1), si et seulement si :

- l'état cible de t est différent de l'état source de t', alors que l'état cible de t' est l'état source lui-même (voir ligne 2),

- l’opération associée à t est de polarité d’envoie, tandis que celle associée à t’ est de polarité de réception (voir ligne 3),

- le type du message envoyé par l’opération associée à la transition t est équivalent à celui du message reçu par l’opération associée à t’ (voir ligne 4),

- le message envoyé par l’opération associée à t peut comprendre plusieurs éléments du type Type(Message(Operation(t))), (voir ligne 5), alors que le message attendu par l’opération associée à t’ ne peut comprendre qu’un seul élément de type Type(Message(Operation(t’))) (voir ligne 6).

Dans le cas où l’expression logique du patron One-to-One^{+} (présentée ci-dessus) est évaluée à vrai pour les transistion t et t’ sortantes de e et e’ (soit (e, e’) le couple d’états courants), la fonction EvaluateLogicExpressions retourne l’incompatibilité i = <T, T’, P> sous la forme suivante :

- T comprend un seul élément T[0], tel que T[0] = t

- T’ comprend un seul élément T’[0], tel que T’[0] = t’

- P = One-to-One^{+}

Ainsi, le nouveau couple d’états courants est déterminé comme suit : Suivant(i) = (Target(i.T[0]), Target(i.T’[0])). C'est-à-dire nouveau couple d’états courants sont les états destinataires de T[0] et de T’[0].

4.3.2 Exclusion mutuelle et complétude

Dans ce qui suit, nous discutons de l’exclusion mutuelle des expressions booléennes présentées dans la section précédente, ainsi que de la complétude de chacune d’expressions par rapport au patron d’incompatibilité qu’elle modélise. Cependant, nous n’étudions pas la complétude de l’ensemble des expressions proposées dans le sens où elles couvrent toutes les incompatibilités structurelles et comportementales qui peuvent se manifester entre deux services. Des incompatibilités telles que :

- l’exécution de plusieurs opérations différentes deux à deux dans service versus l’exécution de plusieurs autres opérations différentes deux à deux d’un autre service,

- la présence d’une opération dans un service et l’absence de son opération correspondante dans un autre service,

ne font pas partie de l'ensemble des patrons d'incompatibilités que nous proposons. L'une de perspectives de ce travail est de formaliser les expressions qui modélise ces patrons d'incompatibilités.

Exclusion Mutuelle

Dans cette partie, nous démontrons qu'il est impossible que deux ou plusieurs incompatibilités soient associées à deux patrons différents. Pour vérifier l'exclusion mutuelle des différentes expressions discutées à la section précédente, il suffit de démontrer qu'il est impossible que deux expressions soient validées à la fois sur les transitions sortantes d'une paire d'états.

Nous nous appuyons sur l'operateur XOR, pour démontrer l'exclusion mutuelle d'expressions. Soit E_1 et E_2 deux expressions modélisant deux patrons des incompatibilités différentes, E_1 XOR $E_2 \Leftrightarrow E_1 = \neg E_2$, c'est-à-dire :

- Si E_1 est vraie, alors E_2 est fausse
- Si E_2 est vraie, alors E_1 est fausse

D'après cela, nous devons démonter les 11 assertions en-dessous :

[1] $\big((\text{One} - \text{to} - \text{One})\ \text{XOR}\ (\text{One} - \text{to} - \text{Many})\big)$
[2] $\big((\text{One} - \text{to} - \text{One})\ \text{XOR}\ (\text{Many} - \text{to} - \text{One})\big)$
[3] $\big((\text{One} - \text{to} - \text{One})\ \text{XOR}\ (\text{One}^{+} - \text{to} - \text{One})\big)$
[4] $\big((\text{One} - \text{to} - \text{One})\ \text{XOR}\ (\text{One} - \text{to} - \text{One}^{+})\big)$
[5] $\big((\text{One} - \text{to} - \text{Many})\ \text{XOR}\ (\text{Many} - \text{to} - \text{One})\big)$
[6] $\big((\text{One} - \text{to} - \text{Many})\ \text{XOR}\ (\text{One}^{+} - \text{to} - \text{One})\big)$
[7] $\big((\text{One} - \text{to} - \text{Many})\ \text{XOR}\ (\text{One} - \text{to} - \text{One}^{+})\big)$
[8] $\big((\text{Many} - \text{to} - \text{One})\ \text{XOR}\ (\text{One}^{+} - \text{to} - \text{One})\big)$
[9] $\big((\text{Many} - \text{to} - \text{One})\ \text{XOR}\ (\text{One} - \text{to} - \text{One}^{+})\big)$
[10] $\big((\text{One}^{+} - \text{to} - \text{One})\ \text{XOR}\ (\text{One} - \text{to} - \text{One}^{+})\big)$
[11] $\big((\text{One} - \text{to} - \text{One}^{+})\ \text{XOR}\ (?\,(\text{One}) - \text{to} - \text{One})\big)$

Démonstration de [1] :

Soient I et I' deux automates modélisant deux interfaces différentes, et (e, e') une paire d'états tels que e dans I et e' dans I'. Soient t_1 et t'_1 deux transitions telles $t_1 \in$ Outbounds(e) et $t'_1 \in$ Outbounds(e'), et tr une trace telle que, tr = $[t'_1, t'_2, \dots, t'_n] \in$ Traces(e', e_n).

Raisonnons par l'absurde. Supposons que l'expression de l'incompatibilité One-to-One, noté E_1 dans la suite de cette démonstration, est évaluée à vrai pour la paire d'états (e, e') entre les transitions t_1 et t'_1, et que l'expression One-to-Many, notée E_2 dans la suite de cette démonstration, est évaluée à vrai pour la même paire d'états (e, e') entre la transition t_1 d'une part et la trace $tr = [\, t'_1, t'_2, \dots, t'_n]$ d'une autre part.

Supposons que $E_1 \wedge E_2$, avec :

E_1 :

1. $\text{Name}(\text{Message}(\text{Operation}(t_1)) = \text{Name}(\text{Message}(\text{Operation}(t'_1)) \wedge$
2. $\text{Polarity}(\text{Operation}(t1)) = \overline{\text{Polarity}(\text{Operation}(t'_1))} \wedge$
3. $\mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1})) \not\equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1}))$

et E_2 :

1. $\text{Polarity}\big(\text{Operation}(t1)\big) = ! \ \wedge \ (\bigwedge_{i=1}^{n} \text{Polarity}(\text{Operation}(t'_1) = ?) \wedge$
2. $\neg(\bigvee_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1}))) \equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1})))) \wedge$
3. $\text{Type}(\text{Message}(\text{Operation}(t_1))) \equiv \bigcup_{i=1}^{n}(\text{Type}(\text{Message}(\text{Operation}(t'_i))))$

$E_1 \wedge E_2$ est équivalente à :

1. $\text{Name}(\text{Message}(\text{Operation}(t_1)) = \text{Name}(\text{Message}(\text{Operation}(t'_1))) \wedge$
2. $\text{Polarity}(\text{Operation}(t1)) = \overline{\text{Polarity}(\text{Operation}(t'_1))} \wedge$
3. $\mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1})) \not\equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1})) \wedge$
4. $\text{Polarity}\big(\text{Operation}(t1)\big) = ! \ \wedge \ (\bigwedge_{i=1}^{n} \text{Polarity}(\text{Operation}(t'_1) = ?) \wedge$
5. $\neg(\bigvee_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1}))) \equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1})))) \wedge$
6. $\text{Type}(\text{Message}(\text{Operation}(t_1))) \equiv \bigcup_{i=1}^{n}(\text{Type}(\text{Message}(\text{Operation}(t'_i))))$

Après avoir développé la ligne 5, $E_1 \wedge E_2$ est équivalente à :

1. $\text{Name}(\text{Message}(\text{Operation}(t_1)) = \text{Name}(\text{Message}(\text{Operation}(t'_1))) \wedge$
2. $\text{Polarity}(\text{Operation}(t1)) = \overline{\text{Polarity}(\text{Operation}(t'_1))} \wedge$
3. $\mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1})) \not\equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1})) \wedge$
4. $\text{Polarity}\big(\text{Operation}(t1)\big) = ! \ \wedge \ (\bigwedge_{i=1}^{n} \text{Polarity}(\text{Operation}(t'_1) = ?) \wedge$
5. $\mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1})) \equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1}))) \wedge \dots \wedge \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1})) \equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_n})))$
6. $\text{Type}(\text{Message}(\text{Operation}(t_1))) \equiv \bigcup_{i=1}^{n}(\text{Type}(\text{Message}(\text{Operation}(t'_i))))$

D'après les lignes 3 et 5, nous avons :

$\mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1})) \not\equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1})) \wedge$
$\mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t1})) \equiv \mathbf{Type}(\mathbf{Message}(\mathbf{Operation}(\mathbf{t'_1})))$

Ce qui est une contradiction, donc $E_1 \wedge E_2$ est fausse. Ainsi, l'hypothèse de départ est fausse,

et il est impossible qu'une expression $\mathrm{One}-\mathrm{to}-\mathrm{One}$ et une autre $\mathrm{One}-\mathrm{to}-\mathrm{Many}$ soient vraies en même temps pour une même paire d'états. On en déduit que: (One-to-One) XOR (One-to-Many).

Les démonstrations des assertions [2] à [11] suivent le même principe de raisonnement par l'absurde que celui mené pour démontrer [1].

Complétude de chaque expression

D'après Gödel [GÖD29], on dit d'un objet mathématique qu'il est *complet pour exprimer* que rien ne peut lui être ajouté, et ce dans un sens qu'il faut préciser dans chaque contexte. Dans le cas contraire, on parle *d'incomplétude*.

Dans notre contexte, une expression de détection d'un patron d'incompatibilité est dite complète, quand elle couvre tous les cas de l'incompatibilité qu'elle modélise. Ainsi, chacune des expressions de détection des incompatibilités présentées à la section 4.3.1, est complète. Par exemple, l'expression $\mathrm{One}-\mathrm{to}-\mathrm{One}$ est vérifiée pour toutes les incompatibilités résultantes d'une différence structurelle entre deux messages échangés par deux opérations. Aussi, l'expression $\mathrm{One}-\mathrm{to}-\mathrm{Many}$ est vérifiée pour toutes les incompatibilités dues à une correspondance entre une opération d'un service, et plusieurs opérations d'un autre service.

Pour démontrer la complétude de chacune de nos expressions, nous nous sommes appuyés sur le principe du raisonnement par l'absurde.

Preuve de la complétude de l'expression One-to-One:

Supposons qu'il existe une incompatibilité de type One-to-One (noté E) entre deux transitions t et t' pour laquelle l'expression booléenne modélisant l'incompatibilité de type One-to-One est non-vérifié. Cela veut dire que sa négation (¬E) est vérifiée :

1. $\neg[\ \mathrm{Name}(\mathrm{Message}(\mathrm{Operation}(t))) = \mathrm{Name}(\mathrm{Message}(\mathrm{Operation}(t'))) \wedge$
2. $\mathrm{Polarity}(\mathrm{Operation}(t)) = \overline{\mathrm{Polarity}(\mathrm{Operation}(t'))} \wedge$
3. $\mathrm{Type}(\mathrm{Message}(\mathrm{Operation}(t))) \not\equiv \mathrm{Type}(\mathrm{Message}(\mathrm{Operation}(t')))\ ]$

Ce qui est équivalent à :

1. $\mathrm{Name}(\mathrm{Message}(\mathrm{Operation}(t))) \neq \mathrm{Name}(\mathrm{Message}(\mathrm{Operation}(t')))\ \vee$
2. $\mathrm{Polarity}(\mathrm{Operation}(t)) = \mathrm{Polarity}(\mathrm{Operation}(t'))\ \vee$
3. $\mathrm{Type}(\mathrm{Message}(\mathrm{Operation}(t))) \equiv \mathrm{Type}(\mathrm{Message}(\mathrm{Operation}(t')))$

Nous allons démontrer que cette expression s'évalue à faux. Pour ce faire, il faut

démonter que les assertions des lignes 1, 2, et 3 sont fausses.

La validité de l'assertion de la ligne 1 signifie que les noms de messages envoyés ou reçus par les opérations associées à t et t' sont de noms différents. Or, par définition, l'incompatibilité de type One-to-One survient entre deux messages dont les noms sont les mêmes. Par suite, l'assertion de la ligne 1 ne peut pas être vraie.

La validité de l'assertion de la ligne 2 signifie que les opérations associées à t et t' sont de même polarité, cela signifie qu'ils ne peuvent pas interagir l'un avec l'autre, alors aucun type d'incompatibilité ne peuvent avoir lieu entre elles. Par suite, l'assertion de la ligne 2 ne peut pas être vraie.

La validité de l'assertion de la ligne 3 signifie que les deux messages ont deux types structurels équivalents, ce qui contredit le principe de l'incompatibilité de type One-to-One. Par suite, l'assertion de la ligne 3 ne peut pas être vraie ne peut pas aussi être vraie.

Aucune de ces trois conditions n'est vraie, alors ¬E est fausse, par suite E est vraie. Ce qui permet de conclure que l'expression E est complète pour toute incompatibilité de type One-to-One.

Les preuves de la complétude d'expressions : One-to-Many, Many-to-One, One^{+}-to-One, One-to-One^{+} suivent le même principe de raisonnement par l'absurde que celui mené pour la preuve de la complétude de l'expression One-to-One.

4.3.3 Algorithme de détection d'incompatibilités

Dans cette section, nous détaillons l'algorithme de détection des incompatibilités entre deux interfaces modélisées en automates. Pour ce faire, l'algorithme parcourt de manière synchrone les deux automates. Il commence avec le couple correspondant aux états initiaux des deux interfaces, et il arrête son exécution dès que tous les états de l'une des deux interfaces ont été visités.

Dans l'algorithme 4.4.2, I et I' représentent les interfaces à comparer, données en paramètres d'entrée à l'algorithme. Inc représente la liste des incompatibilités détectées entre I et I'. Inc est retournée en paramètre de sortie de l'algorithme. Les lignes 1 et 2, déclarent les variables intermédiaires e et e' de type State, et i de type Incompatibility. Ces variables sont utilisés en tant que variables locaux dans l'algorithme. La ligne 3 affecte l'état initial de I InitialState(I) au variable e. Ainsi, la ligne 4 affecte l'état initial de I' InitialState(I') au variable e'. (e,e') est considéré en tant que le couple d'états courants.

Algorithme 4.4. 1 : Algorithme de détection des incompatibilités

```
Input   I, I': Interface                                  {Paramètres d'entrée : les deux interfaces à comparer}
Output  Inc : [Incompatibility]                           {Paramètres de sortie : Liste des incompatibilités}
  Begin
    1.  e, e' : State                                     {variables intermédiaires de type state}
    2.  i : Incompatibility                               {variable intermédiaire de type Incompatibility}
    3.  e ← intialState(I)                                {Affectation de l'état initial de I à e}
    4.  e' ← intialState(I')                              {Affectation de l'état initial de I' à e'}
    5.  While e ≠ FinalState(I) or e' ≠ FinalState(I') do {Condition à verifier}
    6.  i ← EvaluateLogicExpressions(e,e')                {Evaluation des expressions logiques}
    7.  Inc ← Add-Incompatibility (Inc, i)                {Ajouter l'incompatibilité détectée à Inc}
    8.  (e, e') ← Suivant (i)                             {nouveau couple d'états courants}
    9.  EndWhile
  End
```

Dans la ligne 5, la condition vérifie si tous les états de l'une des deux interfaces on été visités. En effet, les états d'une interface I sont visités si l'état courant e de I est son état final (e = Final state(I)). Tant que la condition est vraie, les expressions qui modélisent les patrons des incompatibilités (données dans la section 4.3.1) sont évaluées sur le couple d'états courants (e, e'), et soit i l'incompatibilité détectée par la fonction d'évaluation des expressions logiques EvaluateLogicExpression (voir la ligne 6). Dans la ligne 7, l'incompatibilité détectée i est ajoutée à la liste Inc des incompatibilités à retourner par l'algorithme.

Dans la ligne 8, le nouveau couple d'états courant est déterminé en fonction de l'incompatibilité détectée (voir la définition de la fonction Suivant donné dans la section 4.3.1 pour chaque patron d'incompatibilités).

4.4 Résolution des incompatibilités

Dans cette section, nous introduisons un jeu d'opérateurs d'adaptation prédéfinis exprimés sous forme d'automates configurables. En effet, pour chaque patron d'incompatibilité présenté à la section précédente, nous proposons un patron de résolution correspondant. Celui-ci est configuré et utilisé par l'algorithme de construction des adaptateurs des interactions entre services. Un adaptateur des interactions entre deux services est une composition de l'ensemble des opérateurs de résolution qui correspondent aux incompatibilités détectées entre les deux interfaces de ces services. Dans la section 4.4.1, nous décrivons le principe de la résolution à base des opérateurs. La section 4.4.2 propose une caractérisation du jeu d'opérateurs proposés. Enfin, la section 4.4.3 détaille l'algorithme proposé pour la génération des adaptateurs.

4.4.1 Résolution à base des operateurs

Comme nous l'avons illustré auparavant, le principe de l'adaptation d'interactions entre deux services, consiste à mettre en place un composant automatiquement généré, nommé adaptateur. L'automatisation de la génération de l'adaptateur s'appuie sur la configuration et la composition d'opérateurs de résolution.

Un opérateur de résolution peut être considéré comme un adaptateur élémentaire. Son rôle se limite à résoudre une incompatibilité spécifique. Un opérateur typique consiste à consommer un ou plusieurs messages provenant d'un service émetteur, les manipuler à l'aide d'une fonction de transformation prédéfinie, et enfin les transmettre au service destinataire.

Un opérateur est exécuté à la réception du premier message de la part du service émetteur (au cas où il y en a plusieurs), et son exécution se termine à la transmission du dernier message au service destinataire. Pendant son exécution, un opérateur passe dans des phases différentes, telles que : consommation, attente, transformation, et transmission. Ainsi, l'exécution d'une phase dépend de l'exécution de la précédente. Par exemple, la transformation d'un message ne peut pas être accomplie avant sa consommation. Aussi, la transmission d'un message ne peut pas avoir lieu avant sa transformation, et ainsi de suite.

Lorsqu'il s'agit de modéliser les opérateurs, il est indispensable de s'appuyer sur un langage qui permet d'exprimer les différentes phases dans lesquelles un opérateur peut passer pendant son exécution. Pour ces raisons et en harmonisation avec le modèle utilisé pour la représentation de l'interface de services, nous avons choisi de modéliser les patrons de résolution à l'aide des automates.

Dans cette modélisation, les états de l'automate représentent l'ensemble des phases dans lesquelles un opérateur peut passer lors de ses interactions avec deux services consommateur et fournisseur. Les transitions représentent les actions *observables* mais aussi celles *inobservables* d'opérateurs.

Les actions observables décrivent le fonctionnement et le comportement de l'opérateur vis-à-vis de ses partenaires (services consommateur et fournisseur). Une action *observable* est, soit la consommation d'un message de la part du service émetteur, soit la transmission d'un message au service destinataire.

Les actions inobservables (noté ψ) décrivent le fonctionnement interne de l'opérateur, tel que : la manipulation et la transformation de messages. Ces actions sont accomplies de façon transparente pour les partenaires de l'adaptation réalisée par l'operateur.

Une transition étiquetée par une action *observable*, notée <a, ?m , a'>, qui s'écrit également $a \xrightarrow{?a} a'$, modélise le passage de l'opérateur de l'état a à l'état a' suite à l'action de consommation du message m provenant du service émetteur.

Une transition étiquetée par une action *inobservable*, notée <a, ψ, a'>, qui s'écrit également $a \xrightarrow{\psi} a'$, modélise le passage de l'operateur de l'état a à l'état a' suite à l'exécution de l'action interne ψ.

4.4.2 Opérateurs d'adaptation

Dans cette section, nous présentons et caractérisons l'ensemble du jeu d'opérateurs proposés.

Mise en correspondance

Nous introduisons l'opérateur de Mise En Correspondance (MEC) pour résoudre les incompatibilités du patron One-to-One.

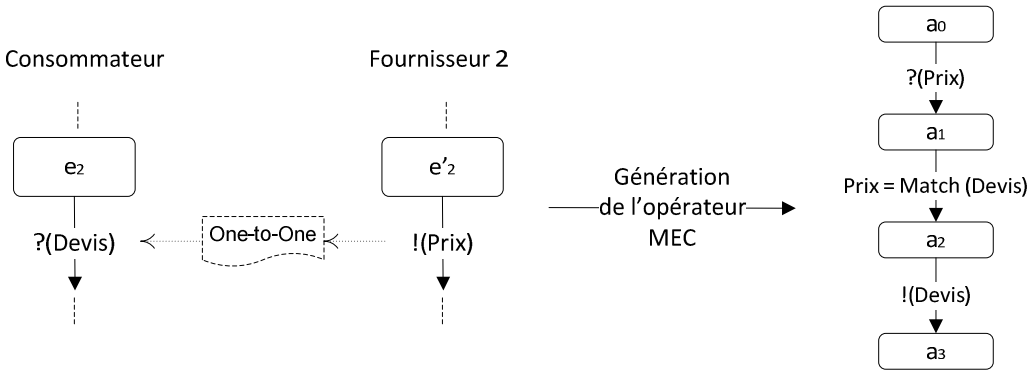

Figure 4.4 Illustration de l'opérateur MEC

Dans Figure 4.4, une incompatibilité de type One-to-One est localisée entre les transitions étiquetées par !(Prix) de l'interface du service Fournisseur 2 et la transition étiquetée par ?(Devis) de l'interface du service Consommateur. Pour résoudre cette incompatibilité, l'operateur de mise en correspondance se place en adaptateur des interactions entre les deux services. Son exécution comporte les actions suivantes :

- La consommation du message Prix provenant du service Fournisseur 2.
- La transformation du type structurel du message consommé selon le type structurel requis

par le service Consommateur.

- Enfin, la transmission au service consommateur du message Prix, obtenu par transformation du message consommé.

L'interface qui décrit le comportement et le fonctionnement de l'opérateur MEC est formalisée par l'automate dont les transitions sont (voir la figure 4.4) :

- $<a_0$, ?(Prix), $a_1>$, représente une transition étiquetée par l'action *observable* ?(Prix) dont l'exécution est déclenchée par la réception du message Prix provenant du partenaire Fournisseur 2. Cette transition est la transition *réciproque* (voir la définition 3.3.2 dans le chapitre 3) de la transition étiquetée par !(Prix) dans l'interface du service Fournisseur 2 (voir la figure 4.4).

- $<a_1$, Devis = Match(Prix), $a_2>$, représente une transition étiquetée par une action *inobservable* Devis = Match(Prix). L'exécution de ce dernier permet transformer la structure de l'instance du message consommé Prix en une structure qui correspond à celle du message Devis attendu par le service Consommateur, et cela à l'aide de la fonction de transformation structurelle Match.

- $<a_2$, !(Devis), $a_3>$, représente une transition étiquetée par l'action *observable* !(Devis) dont l'exécution se traduit par l'envoi au service Consommateur du message Devis, obtenu par la transformation ci-dessus. Cette transition est une transition *réciproque* de la transition annoté par ?(Devis) dans l'interface du service Consommateur (voir la figure 4.4).

La méthode 4.4.1, décrit la procédure de construction et de configuration de l'automate de l'operateur MEC destiné à résoudre les incompatibilités de type One-to-One. Avant de la détailler, nous spécifions les types abstraits, les fonctions et des notations utilisés dans cette méthode.

Soit i, tel que i = <T, T', One-to-One>, une incompatibilité de type One-to-One. Les notations suivantes sont utilisées sur i:

- i.T[0] : représente la transition de polarité ! impliquée par l'incompatibilité i du coté du service émetteur. Dans la figure 4.4, i.T[0] est la transition étiquetée par !(Prix) dans l'interface du service Fournisseur 2,

- i.T'[0] : représente la transition de polarité ? impliquée par l'incompatibilité i du coté du service récepteur. Dans la figure 4.4, i.T'[0] est la transition étiquetée par ?(Prix) dans

l'interface du service Consommateur.

Nous désignons par Operator le type abstrait qui modélise un opérateur. Il comprend une liste des états et une liste des transitions (en faisant abstractions des autres détails tels que, état initial, états finaux, etc.):

type Operator ⟨

a : [State]	{Liste des états}
t : [Transition]	{Liste des transitions}

⟩

E étant un élément du type Operator, les notations suivantes sont utilisées sur E :

- E.a[j] : represente le j^{ième} état de l'opérateur E.
- E.t[j] : représente la j^{ième} transition de l'opérateur E.

La fonction New est également utilisée sur E. Nous la spécifions comme suit :

New : Operator x integer x integer ⟶ Operator

{New(O, m, n) est l'opérateur dont le nombre des états est initialisé à m, et le nombre de ses transitions est initialisé à n}

Le type abstrait TransformationFunction est spécifié comme suit :

type TransformationFuction : {XSLT }

{est le type qui modélise les fonctions de transformation codées en XSLT}

Méthode 4.4.1 : Construction et configuration de l'automate de l'operateur MEC

1. **Input** i : Incompatibility, Match : TransformationFunction
2. **Output** E : Operator
3. **Begin**
4. E ← New E (a [4], t [3])
5. E.t[0] ← Transition(E.a[0], $\overline{\text{Operation}(\text{i.T}[0])}$,E.a[1])
6. E.t[1] ← Transition(E.a[1], Message(Operation(i.T'[0])) = Match(Message(Operation(i.T[0]))), E.a[2])
7. E.t[2] ← Transition(E.a[2], $\overline{\text{Operation}(\text{i.T}'[0])}$, E.a[3])
8. **End**

Dans la ligne 1 de la méthode 4.4.1, i et Match sont donnés en paramètres d'entrée. i représente une incompatibilité de type One-to-One, alors que Match représente la fonction de transformation structurel permettant d'adapter la structure du message consommé, en une

structure qui correspond à celle du message attendu par le service récepteur.

Dans la ligne 2, E représente l'opérateur MEC que la méthode est destinée à construire et à configurer. Etant donné qu'un opérateur MEC comprend quatre états et trois transitions (voir la figure 4.4), la ligne 4 initialise le nombre des états de l'opérateur E à 4 et le nombre de ses transitions à 3, et cala à l'aide de la fonction New. La ligne 5, spécifie la première transition de l'opérateur E.t[0], destinée à consommer le message envoyé par le service émetteur. Cette transition est spécifiée de façon qu'elle soit la *réciproque* à la transition i.T[0]. Dans la figure 4.4, E.t[0] correspond à la transition étiquetée par ?(Prix) de l'operateur MEC, et i.T[0] correspond à la transition étiquetée par !(Prix) dans l'interface du service Fournisseur 2. La ligne 6, spécifie la transition E.t[1], dont le déclenchement applique la fonction de transformation Match sur le message consommé ci-dessus. Dans la figure 4.4, E.t[1] correspond à la transition étiquetée par Devis=Match(Prix) de l'opérateur MEC. La ligne 7, spécifie la transition E.t[2] destinée à transmettre au service récepteur le message obtenu par la transformation ci-dessus. Cette transition est spécifiée de façon qu'elle soit la *réciproque* à la transition i.T'[0]. Dans la figure 4.4, E.t[2] correspond à la transition étiquetée par !(Devis) de l'operateur MEC, et i.T'[0] correspond à la transition étiquetée par ?(Devis) dans l'interface du service Consommateur.

Fraction

Nous associons l'operateur de fraction au patron d'incompatibilité du patron One-to-Many

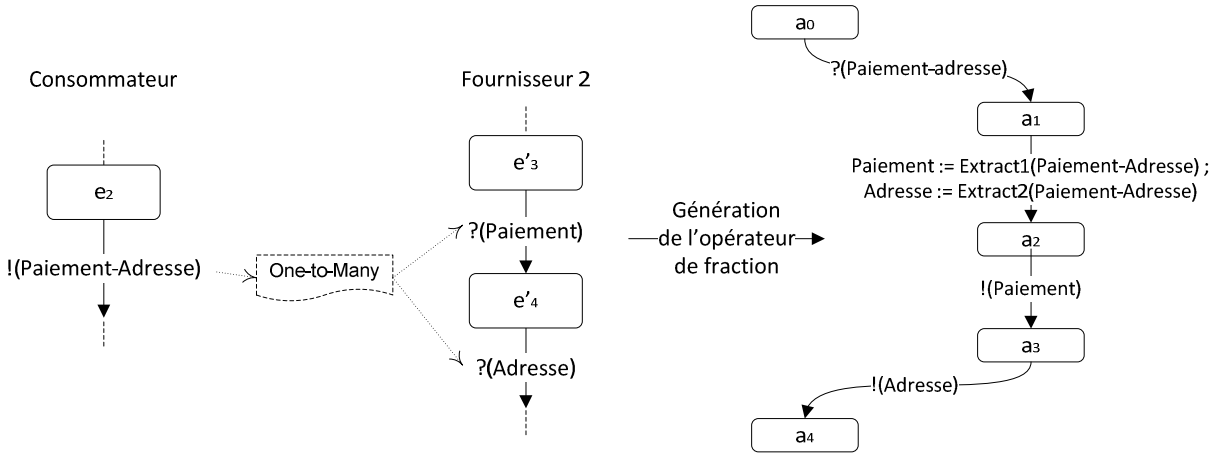

Figure 4.5 Illustration de l'operateur de fraction

Dans Figure 4.5, une incompatibilité de type One-to-Many est localisée entre la transition

étiquetée par l'opération !(Paiement-Adresse) de l'interface du service Consommateur, et les transitions étiquetées par les opérations ?(Paiement) et ?(Adresse) de l'interface du service Fournisseur 2. Pour résoudre cette incompatibilité, l'operateur de Fraction comprend les actions suivantes :

- La consommation du message Paiement-Adresse provenant du service Consommateur
- La fraction du message consommé en deux sous messages : Paiement et Adresse
- La transmission du message Paiement suivie de la transmission du message Adresse.

L'interface qui décrit le comportement et le fonctionnement de l'operateur Fraction est formalisée par l'automate dont les transitions sont :

- $<a_0$, ?(Paiment-Adresse), $a_1>$, représente une transition étiquetée par l'action *observable* ?(Paiment-Adresse) dont l'exécution est déclenchée par la réception du message Paiment-Adresse provenant du service Consommateur.
- $<a_1$, Paiement = Extract1(Paiment-Adresse); Adresse = Extract2(Paiment-Adresse), $a_2>$, représente une transition étiquetée par une action *inobservable* dont l'exécution permet d'extraire à partir du message Paiment-Adresse les deux sous messages Paiement et Adresse.
- $<a_2$, !(Paiement), $a_3>$ (respect. $<a_3$, !(Adresse) , $a_4>$) représente une transition étiquetée par l'action *observable* !(Paiment) (respectivement !(Adresse)) dont l'exécution se traduit par l'envoi au service Fournisseur 2 le message Paiement (respectivement Adresse), obtenu par la transformation ci-dessus.

La méthode 4.4.2, décrit la procédure de construction et de configuration de l'automate de l'operateur Fraction destiné à résoudre les incompatibilités de type One-to-Many.

Dans la ligne 1 de la méthode 4.4.2, i et Frct sont donnés en paramètres d'entrée. i représente une incompatibilité de type One-to-Many, alors que Frct représente la liste des fonctions de transformation, tel que tel que Frct[j] (le $j^{ième}$ élément de la liste) est la fonction de transformation qui permet d'extraire le sous message du type structurel Type(Message(Operation(i.T'[j]))) à partir d'un message du type Type(Message(Operation(i.T[0]))). Dans la ligne 2, Fr représente l'opérateur du type Operator que la méthode est destinée à construire et à configurer. Dans la ligne 4, n représente le nombre des transitions de l'interface du service récepteur, impliquées par l'incompatibilité (dans l'exemple de la figure 4.5, n = 2).

Étant donné qu'un opérateur de Fraction comprend toujours deux états et deux transitions en plus du nombre n (voir la figure 4.5), la ligne 5 initialise le nombre des états de l'opérateur Fr à n + 3 et le nombre de ses transitions à n + 2, à l'aide de la fonction New. La ligne 6 spécifient la transition Fr.t[0] de l'opérateur Fr de manière qu'elle soit la transition réciproque de i.T[0], ce qui lui permet de consommer le message envoyé par l'opération associée à i.T[0]. Dans l'exemple de la figure 4.5, Fr.t[0] correspond à la transition étiquetée par ?(Paiement-Adresse) de l'opérateur de Fraction et i.T[0] correspond à la transistion étiquetée par !(Paiment-Adresse) du Consommateur. La ligne 7 spécifie la transition Fr.t[1] dont le déclenchement permet d'exécuter les fonctions de transformation permettant d'extraire les n sous messages à partir du message reçu. Dans la figure 4.5, Fr.t[1] correspond à la transistion étiquetée par Paiement = Extract1(Paiment-Adresse); Adresse = Extract2(Paiment-Adresse) de l'opérateur de Fraction. Les lignes 5, 6, 7, 8 et 9 spécifient les transitions chargées d'envoyer au service destinataire les messages obtenues par les transformations ci-dessus. Dans la figure 4.5, ces transistion sont celles étiquetées par !(Paiement) et !(Adresse).

Méthode 4.4.2 : Construction et configuration de l'automate de l'operateur de fraction

1. **Input** i : Incompatibility, Ext : [TransformationFunction]
2. **Output** Fr : Operator
3. **Begin**
4. n ← Longueur(i.T')
5. Fr ← New Fr (a [n + 3], t [n +2])
6. Fr.t[0] ← Transition (a[0], $\overline{\text{Operation}(\text{i.T}[0])}$, a[1])
7. Fr.t[1] ← Transition (a[1], Message(Operation(i.T'[j])) = Ext[j](Message(Operation(i.T[0])))$_{j=0,\ldots,n}$, a[2])
8. m: integer, m ← 0
9. While m < n do
10. Fr.t[m+2] ← Transition (a[m+2], $\overline{\text{Operation}(\text{i.T}'[\text{m}])}$, a[m+3])
11. m = m+1
12. EndWhile
13. **End**

Fusion

L'opérateur de fusion est associé au patron d'incompatibilité du patron Many-to-One. Dans Figure 4.6, une incompatibilité de type Many-to-One est localisée entre les transitions étiquetées par les opérations ?(Paiement) et ?(Adresse) de l'interface du service Consommateur et la transition étiquetée par l'opération !(Paiement-Adresse) de l'interface du service Fournisseur 2 (l'exemple de cette figure discute le cas inverse de celui présenté dans l'exemple de la figure 4.5).

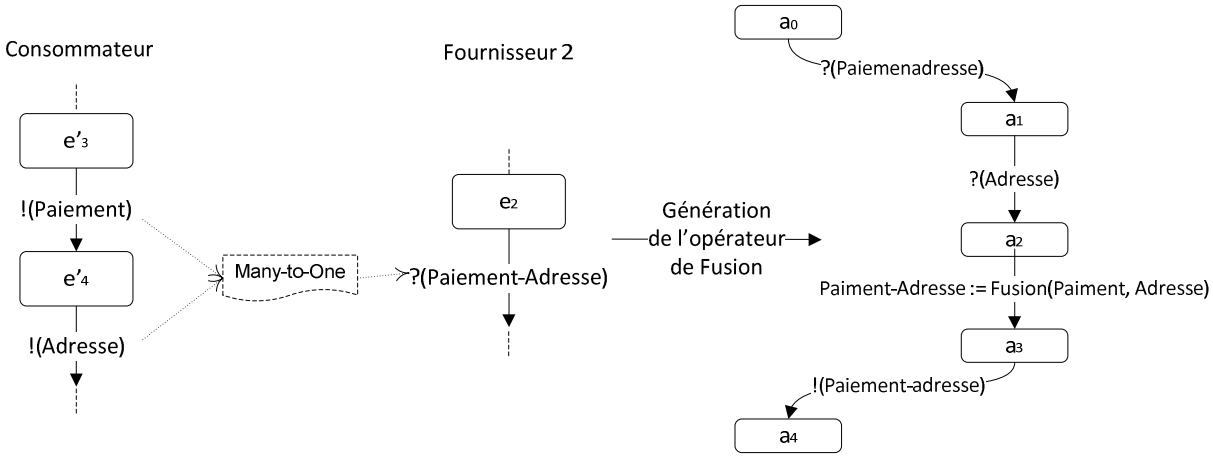

Figure 4.6 Illustration de l'operateur de fusion

Pour résoudre cette incompatibilité, l'operateur de Fusion comprend les actions suivantes :

- La consommation du message Paiement (respectivement du message Adresse) provenant du service Consommateur

- La fusion des deux messages consommés Paiement et Adresse en un seul message Paiement-Adresse

- La transmission du message Paiement-Adresse au service Fournisseur 2.

L'interface qui décrit le comportement et le fonctionnement de l'operateur Fraction est formalisée par l'automate dont les transitions sont :

- $<a_0$, ?(Paiement), $a_1>$ (respect. $<a_1$, ?(Adresse) , $a_2>$) représente une transition étiquetée par l'action *observable* ?(Paiment) (respectivement ?(Adresse)) dont l'exécution se traduit par la consommation du message Paiement (respectivement Adresse), provenant du service Consommateur.

- $<a_2$, Paiement-Adresse = Fusion (Paiment, Adresse), $a_3>$, représente une transition étiquetée par une action *inobservable* dont l'exécution permet de fusionner les deux messages Paiement et Adresse consommés, en un seul message Paiment-Adresse.

- $<a_3$, !(Paiment-Adresse), $a_4>$, représente une transition étiquetée par l'action *observable* !(Paiment-Adresse) dont l'exécution déclenchée l'envoi du message Paiment-Adresse au service Fournisseur.

La méthode 4.4.3, décrit la procédure de construction et de configuration de l'automate de l'operateur Fusion destiné à résoudre les incompatibilités de type Many-to-One.

Dans la ligne 1 de la méthode 4.4.3, i et Fus sont donnés en paramètres d'entrée. i représente une incompatibilité de type One-to-Many, alors que Fus représente la fonction de transformation qui permet la fusion des n messages dont le $j^{ème}$ message est du type structurel Type(Message(Operation(i.T[j]))) en un seul message de type structurel Type(Message(Operation(i.T'[0]))). Dans la ligne 2, Fu représente l'opérateur du type Operator que la méthode est destinée à construire et à configurer. Dans la ligne 4, n représente le nombre des transitions de l'interface du service émetteur, impliquées par l'incompatibilité (dans l'exemple de la figure 4.6, n = 2). La ligne 5 initialise le nombre des états de l'opérateur Fu à n + 3 et le nombre de ses transitions à n + 2. Les lignes 6 à 10 spécifient les n transitions destinées à consommer les n messages provenant du service émetteur. Pour tout j entre 0 et n, la $j^{ème}$ transistion Fu.t[j] est la transition réciproque de i.T[j]. Par exemple, dans la figure 4.6, les deux transitions de l'opérateur, qui sont étiquetées par ?(Paiment) et ?(Adresse) sont les transitions réciproques des deux transitions !(Paiment) et !(Adresse) de l'automate du service Fournisseur. La ligne 11 spécifie la transition étiquetée par l'action interne dont l'exécution déclenche la fonction de transformation permettant de fusionner les messages n sous messages consommés en un seul. La ligne 12 spécifie la dernière transition de l'automate Fu.t[n+1] de manière qu'elle soit la *réciproque* à i.T'[0]. Par exemple, dans la figure 4.6 la dernière transition de l'opérateur, étiquetée par !(Paiment-adresse) est la transition réciproque de ?(Paiment-Adresse) dans l'automate du service Fournisseur.

Méthode 4.4.3 : Construction et configuration de l'automate de l'operateur de Fusion

1. **Input** i : Incompatibility, Fus : TransformationFunction
2. **Output** Fu : Operator
3. **Begin**
4. n ← Longueur(i.T)
5. Fu ← New Fu (a [n + 3], t [n +2])
6. m: integer, m ← 0
7. While m < n do
8. Fu.t[m] ← Transition (Fu.a[m], $\overline{\text{Operation}(\text{i.T}[\text{m}])}$, Fu.a[m+1])
9. m = m+1
10. EndWhile
11. Fu.t[n+1]←Transition(Fu.a[n+1],Message(Operation(i.T'[0]))=Fus(Message(Operation(i.T))), a[n+2])
12. Fu.t[n+2] ← Transition (a[n+2], $\overline{\text{Operation}(\text{i.T}'[0])}$, a[n+3])
13. **End**

Agrégation

L'opérateur d'agrégation est mis en œuvre pour résoudre les incompatibilités du patron One^{+}-to-One.

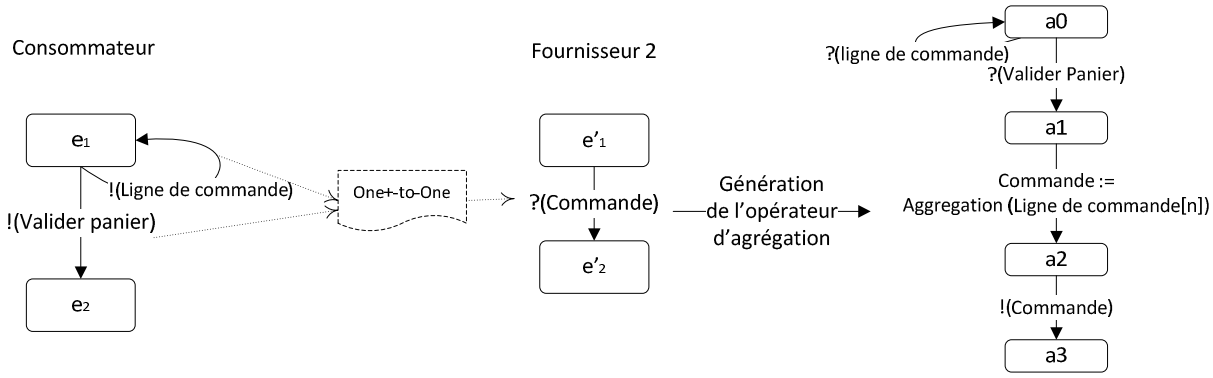

Figure 4.7 Illustration de l'opérateur d'agrégation

Dans la figure 4.7, une incompatibilité de type One^{+}-to-One est localisée entre les transitions étiquetées par !(Ligne de commande) et !(Valider Panier) du service Consommateur d'une part, et la transition étiquetée par ?(Commande) du service Fournisseur 2, d'une autre part. Pour résoudre cette incompatibilité, l'opérateur d'agrégation se comporte comme suit :

- Il consomme les instances du message Ligne de commande provenant du service Fournisseur 2. Chaque consommation se traduit par le déclenchement de la transition <a_0, ?(Ligne de commande) , a_0>.
- Il consomme le message de validation de panier une fois envoyé par le service Consommateur. Cela se traduit par le déclenchement de la transition <a_0, ?(Valider panier), a_1>.
- A l'état a_1, l'opérateur déclenche la transition étiquetée par Commande = Aggregation (Ligne de commande[n])) permettant d'agréger en un seul message les n lignes de commandes consommées.
- Enfin, la transition < a_2, ?(Commande) : Arazon, a_3> est déclenchée, ce qui traduit l'envoi du message Commande au service Fournisseur 2.

La méthode 4.4.4, décrit la procédure de construction et de configuration de l'automate de l'operateur Fraction destiné à résoudre les incompatibilités de type One^{+}-to-One.

Méthode 4.4.4 : Construction et configuration de l'automate de l'operateur d'agrégation

1. **Input** i : Incompatibility, Aggr : TransformationFunction
2. **Output** Ag : Operator
3. **Begin**
4. Ag ← New Ag (a [4], t [4])
5. Ag.t[0] ← Transition(Ag.a[0], $\overline{\text{Operation}(\text{i.T[0]})}$, Ag.a[0])
6. Ag.t[1] ← Transition(Ag.a[0], $\overline{\text{Operation}(\text{i.T[1]})}$, Ag.a[1])
7. Ag.t[2] ← transition(Ag.a[1], Message(Operation(i.T'[0])) = Aggr ([Message(Operation(i.T[0]))]), Ag.a[2])
8. Ag.t[3] ← Transition(a[2], $\overline{\text{Operation}(\text{i.T'[0]})}$, a[3])
9. **End**

Dans la ligne 1 de la méthode 4.4.4, i et Ag sont donnés en paramètres d'entrée. i représente une incompatibilité de type One$^+$-to-One, alors que Ag représente la fonction de transformation qui permet d'agréger les n messages consommées par l'opérateur en un seul. Notons, que les n messages sont de même type Type(Message(Operation(i.T[0]))). Dans la ligne 2, Ag représente l'opérateur du type Operator que la méthode est destinée à construire et à configurer.

La ligne 4 initialise le nombre des états de l'opérateur Ag à 4 ainsi que le nombre de ses transitions. Les lignes 5 et 6 spécifient les deux transitions Ag.t[0] et Ag.t[1] de façon qu'elles soient réciproques aux transistions i.T[0] et i.T[1]. Dans la figure~4.7, Ag.t[0] et Ag.t[1] correspondent aux transitions étiquetés par ?(Ligne de commande) et ?(Valider panier) de l'opérateur d'agrégation, alors que i.T[0] et i.T[1] correspondent aux transitions étiquetées par !(Ligne de commande) et !(Valider panier) de l'interface du service Consommateur. La ligne 7 spécifie la transition Ag.t[2] dont le déclenchement permet d'exécuter l'action interne qui consiste à agréger les messages consommés en un seul message (voir dans la figure 4.7 la transistion étiquetée par Aggreg(Ligne de commande[n]))).

Enfin, la ligne 8 spécifie la dernière transistion de l'opérateur permettant d'envoyer au destinaire le message obtenu par l'agrégation ci-dessus (voir dans la figure 4.7 la transistion étiquetée par !(Commande)).

Itération

L'operateur d'itération est introduit pour résoudre les incompatibilités du patron One-to-One$^+$. L'opérateur consiste à consommer un message sous forme d'une liste de n éléments du même type, provenant d'un service émetteur. Tant que la liste est non-vide, envoyer sa tête à l'action correspondante dans l'interface du service récepteur (voir la figure 4.8).

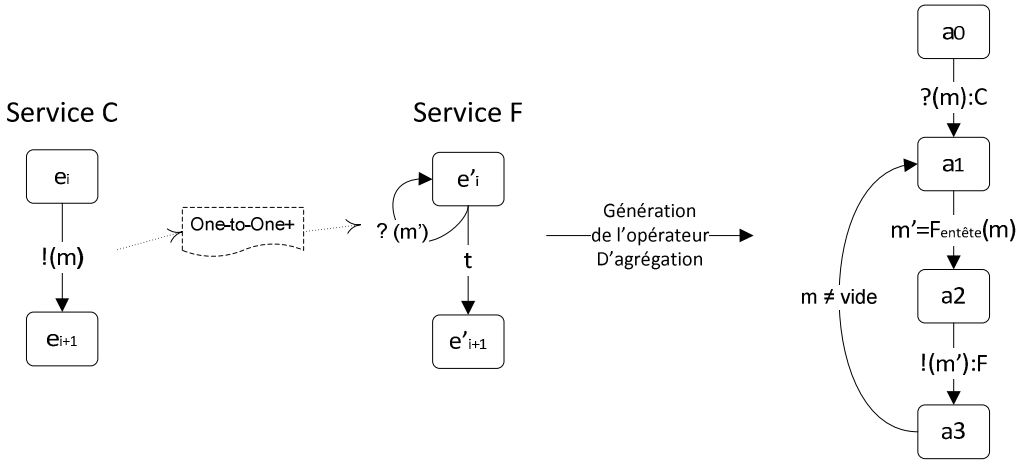

Figure 4.8 Illustration de l'opération d'itération

La méthode 4.4.5, décrit la procédure de construction et de configuration de l'automate de l'operateur d'itération.

Méthode 4.4.5 : Construction et configuration de l'automate de l'operateur d'itération

Input i : Incompatibility, F_{entete} : TransformationFunction
Output It : Operator
Begin

1. It ← New It (a [4], t [4])
2. Message(Operation(T'[0])) = Itr([Message(Operation(T[0]))])
3. It.t[0] ← Transition(It.a[0], $\overline{\text{Operation}(i.T[0])}$, it.a[1])
4. It.t[1] ← Transition(It.a[1], Message(Operation(i.T'[0])) = Itr([Message(Operation(i.T[0]))]), it.a[2])
5. It.t[2] ← Transition(It.a[2], $\overline{\text{Operation}(i.T'[0])}$, It.a[3])
6. It.t[3] ← Transition(It.a[3], m ≠ null , It.a[1])

End

4.4.3 Algorithme de génération d'adaptateurs

Nous détaillons ici l'algorithme de génération d'adaptateurs. Le but de l'algorithme est de construire un automate qui spécifie le fonctionnement et le comportement de l'adaptateur ente deux servies dont les interactions ne peuvent pas aboutir à cause des incompatibilités entre leurs interfaces. Pour ce faire, l'algorithme parcourt la liste des incompatibilités détectées entre les interfaces, et pour chaque incompatibilité de la liste l'operateur d'adaptation correspondant est mis en œuvre. L'algorithme arrête son exécution dès que toutes les incompatibilités de la liste ont été traitées.

Dans l'algorithme 4.4.2, Inc représente la liste des incompatibilités détectées entre les interfaces de deux services, donnée en paramètre d'entrée à l'algorithme. Adapter représente

l'adaptateur retourné en paramètre de sortie de l'algorithme.

Algorithme 4.4.2 : Algorithme de génération d'adaptateurs

Input Inc : [Incompatibility] *{Paramètres d'entrée : Liste des incompatibilités}*
Output Adapter : Operator *{Paramètres de sortie : adaptateur}*
Begin
1. While PremierElement(Inc)
2. Adapter ← AddOperator (Adapter, ConstructOperator (PremierElement(Inc)))
3. EndWhile

End

Dans la ligne 1, la condition vérifie si l'une des incompatibilités de la liste n'a pas encore été traitée. En effet, toutes les incompatibilités de la liste sont traitées si le premier élément de la liste PremierElement(Inc) est null. Tant que le premier élément de la liste n'est pas null, la fonction ConstructOperateur, définie ci-dessous, configure et met en place l'operateur de l'adaptation correspondant à l'incompatibilité PremierElement(In). Ensuite, la fonction AddOperator, définie-ci dessous également, ajouter l'operateur retourné par ConstructOperateur à l'adaptateur Adapter (voir ligne 2).

Les fonctions ConstructOperateur et AddOperator sont spécifiées comme suit :

ConstructOperator : Incompatibility ⟶ Operator

{ConstructOperator(i) construit l'operateur d'adaptation qui correspond à l'incompatibilité i, et cela à l'aide des méthodes 4.4.1, 4.4.2, 4.4.3, 4.4.4, et 4.4.5. données dans la section 4.4. Par exemple, si i est de type One-to-One alors la méthode 4.4.1 est utilisée pour construire l'opérateur MEC destiné à résoudre l'incompatibilité i}

AddOperator : Operator x Operator ⟶ Operator

{AddOperator(O_1, O_2) ajoute à l'operateur O_1, l'operateur O_2. L'addition de l'opérateur O_2 à O_1 se fait sous forme d'une composition séquentielle entre les deux opérateurs, c'est-à-dire : l'état final de O_1 dernier devient l'état initial de O_2 et, l'état final de l'operateur et de garder le reste intact}

4.5 Conclusion

Dans ce chapitre, nous avons présenté notre proposition pour l'adaptation d'interactions de services par détection des incompatibilités et génération des adaptateurs.

La détection des incompatibilités entre deux interfaces modélisées en automates s'appuie sur un ensemble des patrons d'incompatibilités modélisés par le biais des expressions

logiques. La validité d’une expression sur des transitions de deux automates, se traduit par la détection d’une incompatibilité entre les interfaces modélisées par ces automates. Ainsi, l’ensemble des incompatibilités détectées entre les deux interfaces est retourné.

Une fois la phase de détection achevée, la phase suivante consiste à construire un adaptateur destiné à résoudre les incompatibilités détectées. Celui-ci est construit sur la base des opérateurs de résolution primitifs. En effet, à chaque incompatibilité, nous associons un opérateur de résolution. Pour tout élément de l’ensemble des incompatibilités retourné, une méthode est lancée automatiquement pour configurer et mettre en place l’operateur correspondant. La composition de l’ensemble des opérateurs configuré correspond à l’automate qui spécifie le fonctionnement et le comportement de l’adaptateur.

Nous détaillons dans le chapitre suivant la mise en œuvre des opérateurs et des algorithmes formalisés jusqu’ici.

CHAPITRE 5

Mise en œuvre de la proposition avec la technologie CEP

Sommaire

5.1 Introduction

Dans le chapitre précédent, nous avons détaillé notre proposition pour la détection et la résolution des incompatibilités par spécification et génération des adaptateurs en termes d'automates.

L'objectif de ce chapitre est de mettre en place une solution pour générer automatiquement le code exécutable d'un adaptateur à partir de sa spécification en termes d'automates.

Nous avons décidé de mettre en œuvre notre solution avec la technologie de traitement des évènements complexes *CEP* [WDR06]. Le présent chapitre est organisé comme suit. Dans la section 5.2, nous présentons une vue d'ensemble de la technologie *CEP*. La section 5.3 présente une méthode pour la mise en œuvre des opérateurs d'adaptation à l'aide

de la technologie *CEP*. Enfin, la section 5.4 présente le canevas *AdaptTer* pour la mise en œuvre de notre proposition.

5.2 CEP- Traitement des événements complexes

Dans cette section, nous présentons les principes de la technologie *CEP*. D'abord, nous présentons les principes des architectures guidées par les événements. Ensuite, nous présentons un panorama du traitement des évènements complexes. Enfin, nous discutons des éléments de l'infrastructure *CEP*.

5.2.1 Vue d'ensemble

Selon Chandy et coll. [CCC07], un évènement est défini comme un changement significatif d'état d'un système. Par exemple, quand un voyageur règle le montant de sa commande, elle passe de l'état *réservé* à l'état *réglé*. Ensuite, le fournisseur va sur réception de cet évènement déclencher un autre événement qui se traduit par l'envoi de la facture au voyageur.

Une architecture guidée par les événements est une forme d'architecture de médiation. Elle est perçue comme un modèle d'interaction applicative mettant en œuvre des composants logiciels répondant à des sollicitations externes, considérées comme des événements.

Cette architecture repose principalement sur un gestionnaire d'événements disposant des fonctionnalités *d'abonnement* et de *publication*. Un consommateur s'abonne aux événements qu'il souhaite consommer, alors qu'un fournisseur publie des évènements dans le gestionnaire. A la réception d'un événement, le gestionnaire l'évalue pour prendre la décision en fonction de règles prédéfinies. Cette décision peut inclure soit la transmission de l'événement à un consommateur, soit la transformation puis la transmission de celui-ci.

Deux styles de traitement d'événements sont identifiés : *traitement simple* et *traitement complexe*. En cas de *traitement simple*, l'occurrence d'un événement entraine le déclenchement de l'action correspondante sans avoir besoin d'effectuer d'analyse complexe. Par exemple, lorsqu'un détecteur de température saisit une valeur supérieure à un seuil donné, l'alarme est déclenchée.

Cependant, en cas de *traitement complexe*, une analyse sur l'occurrence des événements simples est nécessaire pour détecter l'occurrence d'un événement complexe. Par exemple, si une carte bancaire a été utilisée dans deux endroits différents à un intervalle du temps trop court relativement à la distance entre les deux endroits, le système de supervision doit détecter

une tentative de fraude. La détection de l'occurrence de cet événement nécessite le raisonnement suivant : il s'agit de mesurer la distance entre les deux endroits, ensuite déterminer si le propriétaire de la carte peut se rendre du premier au deuxième endroit pendant l'intervalle du temps dans lequel la carte é été utilisé.

CEP est une technologie majeure dans le traitement, l'analyse, et la gestion des événements complexes dans des environnements dynamiques [WDR06]. *CEP* couple les caractéristiques de la technologie de messagerie avec celles de la gestion de bases de données. Par analogie à la première, *CEP* réagit immédiatement à chaque nouveau fragment de données, considéré comme un événement. Par analogie à la deuxième, *CEP* s'appuie sur un langage de requêtes similaire à *SQL* pour traiter et analyser les données, ainsi que les mettre en corrélation avec des données reçues précédemment.

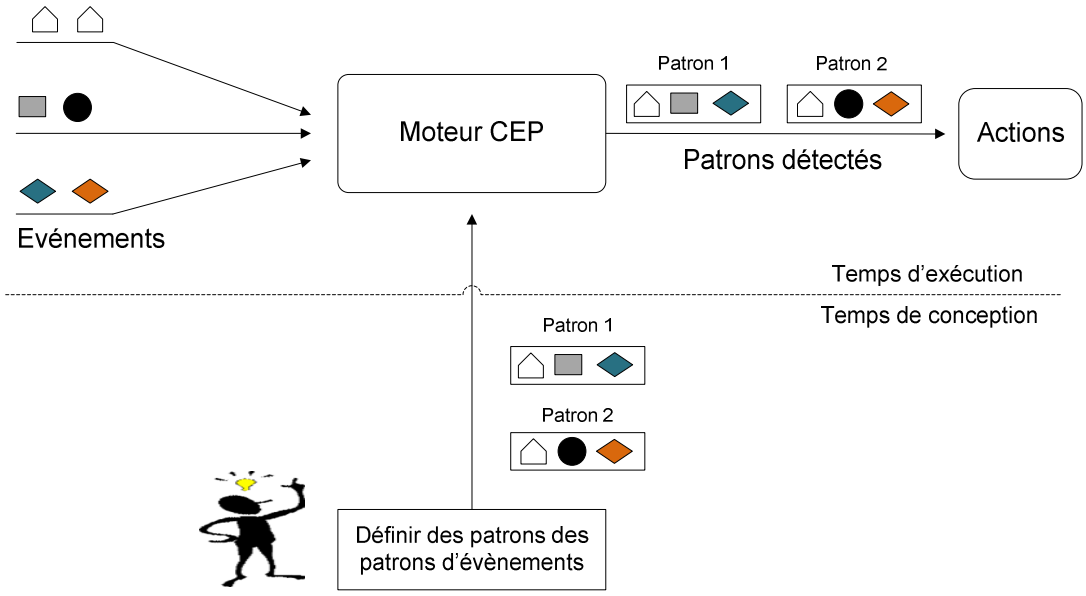

Figure 5.1 Panorama du principe du développement avec la technologie CEP

La technologie *CEP* est principalement utilisée pour développer des applications de traitement d'événements complexes. La figure 5.1 présente un panorama du principe du développement des applications à l'aide de la technologie *CEP* :

- Lors de la phase de conception, le concepteur de l'application spécifie les règles appelées patrons, par le biais d'un langage spécifique, de type *ECA (Event-Condition-Action*). Ensuite, les règles obtenues sont chargées et déployées dans le gestionnaire d'événements.

- Lors de la phase d'exécution, les événements de formats différents, provenant des sources multiples, s'introduisent dans le gestionnaire des événements. Afin de détecter les patrons

prédéfinis, les événements sont traités, corrélés, et analysés sur la base des règles déployées dans le gestionnaire. Une fois la condition d'une règle satisfaite, l'action correspondante est déclenchée. Une action peut se traduire de différentes manières, telles que : la transformation de l'information, le stockage de celle-ci dans une base de données, ou bien l'invocation d'un service Web.

Généralement, une infrastructure *CEP* comporte les éléments suivants :

- *Un langage de traitement continu* (en anglais, *Continuous Processing Language*) qui permet d'écrire des requêtes continues pour implanter les règles de type ECA.
- *Un moteur de traitement des événements complexes (Moteur CEP)* qui héberge et déploie les requêtes continues pour traiter les flux d'événements entrants.
- *Un modèle de données* qui consiste en des flux d'entrée, de sortie, et intermédiaires. Ceux-ci constituent les éléments principaux pour transporter les données à travers le moteur de traitement d'événements.

5.2.2 Langages de traitement continu

Les langages de traitement continu sont utilisés pour spécifier des requêtes permettant de traiter les événements reçus par le moteur *CEP*. Une requête consomme les événements d'un ou plusieurs flux d'événements, analyse ou manipule les événements, puis transmet les résultats obtenus vers une destination. Celle-ci est généralement un autre flux d'évènements (voir la figure 5.2).

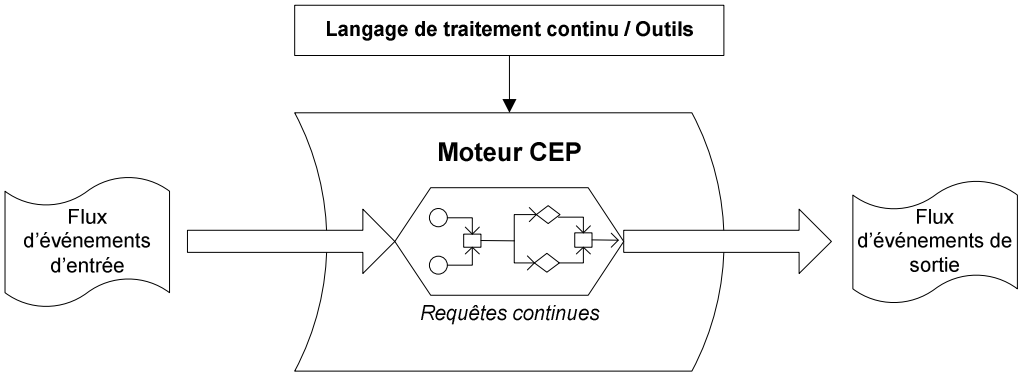

Figure 5.2 Vue d'ensemble de CEP

Les langages de traitement continu sont basés sur *SQL*. Ces langages ont été étendus pour inclure des caractéristiques nécessaires à la manipulation des données en temps réel et du traitement en continu, comme les fenêtres sur les flux de données, les patrons des

événements, et le contrôle temporel de transmission des événements.

L'avantage majeur d'un langage de traitement continu est sa capacité de traiter en continu des données dynamiques. Une requête *SQL* s'exécute généralement une seule fois à chaque fois qu'elle est soumise à un serveur de base de données. Ainsi, à chaque fois que l'utilisateur a besoin de re-exécuter la requête, il doit la soumettre de nouveau auprès du serveur. En revanche, une requête continue est continue : c'est-à-dire qu'une fois soumise au moteur *CEP*, elle est enregistrée pour une exécution continue. Elle reste active indéfiniment afin d'exécuter en continu les événements qui arrivent au fur et à mesure d'une ou de plusieurs sources d'événements.

La plupart des langages de traitement continu fournissent un support natif pour le traitement des événements traduits par l'arrivée de messages *XML*. Ces langages permettent aux développeurs d'intégrer dans le code de leurs requêtes continues des fonctions de traitement offertes par les standards *SQL/XML*, *XQuery*, et *XPath*.

De plus, un langage de traitement continu permet de décrire le schéma de flux d'événements, de spécifier les fonctions de traitement ainsi que leur séquence d'exécution, et de définir comment générer les événements de sortie.

5.2.3 Modèle de données

Alors que les tables et les vues constituent le fondement d'une base de données relationnelle, les flux et les fenêtres d'événements constituent la base sur laquelle les requêtes continues opèrent.

Les flux d'événements sont les éléments principaux pour la transmission de données à travers un moteur *CEP*. Nous distinguons entre deux types de flux : *flux d'entrée* et *flux de sortie*. Selon la figue 5.3, une requête s'abonne à un flux d'entrée pour consommer les événements arrivant dans le moteur *CEP* ; les analyse et les manipule, puis publie les événements ainsi obtenus dans un flux de sortie.

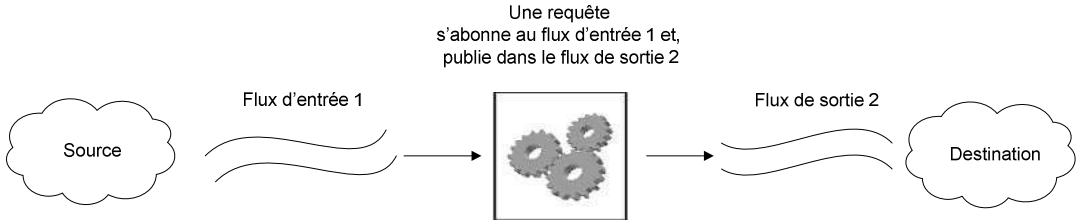

Figure 5.3 Exemple d'abonnement/publication d'une requête aux flux d'entrée/sortie

Comme les tables *SQL*, un *nom* et un *schéma* de données sont associés à chaque flux. Le *schéma* définit les colonnes du flux, ainsi que les types de données de chaque colonne. Une fois qu'une donnée est reçue par un flux, elle apparaît sous la forme d'une ligne, avec une valeur pour chacune des colonnes représentées dans le schéma.

Les flux de données ne peuvent pas maintenir d'état. Par conséquent, seule la dernière ligne d'un flux est accessible aux requêtes abonnées au flux. Ainsi, une ligne est accessible seulement au moment où elle arrive à travers le flux dans le moteur *CEP*.

Toutefois, le principe de traitement *d'une seule ligne en un temps*, n'est pas toujours satisfaisant. En effet, des requêtes peuvent avoir besoin d'opérer sur plusieurs lignes arrivant à travers d'un flux (comme dans le cas d'agrégation de messages). Pour ce faire, les fenêtres d'événements ont été introduites.

Une fenêtre est un ensemble des lignes, elle peut retenir l'état d'un nombre d'événements provenant dans le moteur à travers un flux de données (ou plus qu'un flux de données). En d'autres termes, une fenêtre peut être perçue comme un registre qui permet de stocker les événements arrivés dans le moteur à travers un certain flux. Ce qui permet à une requête abonnée à la fenêtre d'opérer sur l'ensemble des événements stockés.

Dans la section ci-après nous montrons comment utiliser la technologie *CEP* pour mettre en œuvre les adaptateurs que nous avons spécifiés dans le chapitre 4.

5.3 Adaptation à base de CEP

Dans cette section, nous mettons en place une méthode pour la mise en œuvre d'adaptateurs d'interactions de services par le biais de la technologie *CEP*. D'abord, nous présentons les principes généraux de la méthode (voir la section 5.3.1), ensuite nous en décrivons l'architecture conceptuelle (voir la section 5.3.2), et enfin nous spécifions les opérateurs d'adaptation - discutés dans le chapitre précédent - en termes de requêtes continues.

5.3.1 Principe généraux

Comme illustré dans le chapitre précédent, un adaptateur d'interactions entre deux services consiste à consommer un ou plusieurs messages provenant du service émetteur, les manipuler, puis transmettre les messages ainsi obtenus au service destinataire.

D'après une étude[14] expérimentale menée sur les différents langages de développement d'adaptateurs, nous avons constaté que la technologie *CEP* est pertinente pour la mise en œuvre d'adaptateurs. Cette constatation est justifiée par les facilités offertes par un moteur *CEP* en termes d'échanges et transformations de données.

Étant donné qu'un adaptateur d'interactions entre deux services, est spécifié à l'aide d'automates, l'idée consiste donc à transformer les automates en requêtes continues. Nous avons modélisé les actions de consommation et de transmission de messages en termes d'événements. Puis, pour chaque message de consommation, il s'agit de créer un flux d'entrée, puis de créer pour chaque message de transmission, un flux de sortie. Ainsi, la logique d'adaptation de l'adaptateur est encodée en termes de requêtes continues. Une requête s'abonne aux flux d'entrée de messages qu'il souhaite adapter, et publie les messages ainsi obtenus dans les flux de sortie correspondants. Nous associons à un flux d'entrée/sortie le même nom et le même schéma de données que ceux du message qu'il est censé consommer ou transmettre.

Ce procédé permet à un moteur *CEP* d'intercepter les messages échangés entre deux services, de détecter les patrons des incompatibilités, et d'appliquer les solutions d'adaptation correspondantes.

5.3.2 Architecture conceptuelle

La figure 5.4 illustre l'architecture conceptuelle de la mise en œuvre de l'adaptation par le biais de la technologie CEP.

Dans un premier temps, il s'agit de transformer en requêtes continues l'adaptateur spécifié en termes d'automates par le biais du module Automate → Requêtes continues. Cette transformation comprend aussi la création de flux d'entrée/sortie auxquels les requêtes s'abonnent et publient. Ainsi, les requêtes générées sont exécutés en continu dans l'engin CEP en attendant les messages arrivant à travers les flux d'entrée.

Dans un deuxième temps, l'Intercepteur de messages SOAP a pour rôle de superviser et de contrôler l'échange de messages entre les deux services. A la réception d'un message, l'intercepteur le transporte vers le flux d'entrée qui porte le même nom, à travers le composant SOAP → Evénement.

Ainsi, le message publié en tant qu'événement dans le moteur CEP, sera consommé par la

[14] Cette étude a été réalisée dans la cadre de mon séjour scientifique dans l'équipe BPM de l'université de Queensland, à Brisbane-Australie sous la direction de Pr. Marlon Dumas.

requête abonnée au flux à travers lequel le message est arrivé. Puis, le résultat de traitement est publié dans le flux de sortie qui porte le même nom que celui du message obtenu. Une fois le message disponible sur le flux de sortie, il est consommé de nouveau par l'intercepteur de messages SOAP à travers le composant Evénement → SOAP. Enfin, le message est envoyé par l'intercepteur au destinataire.

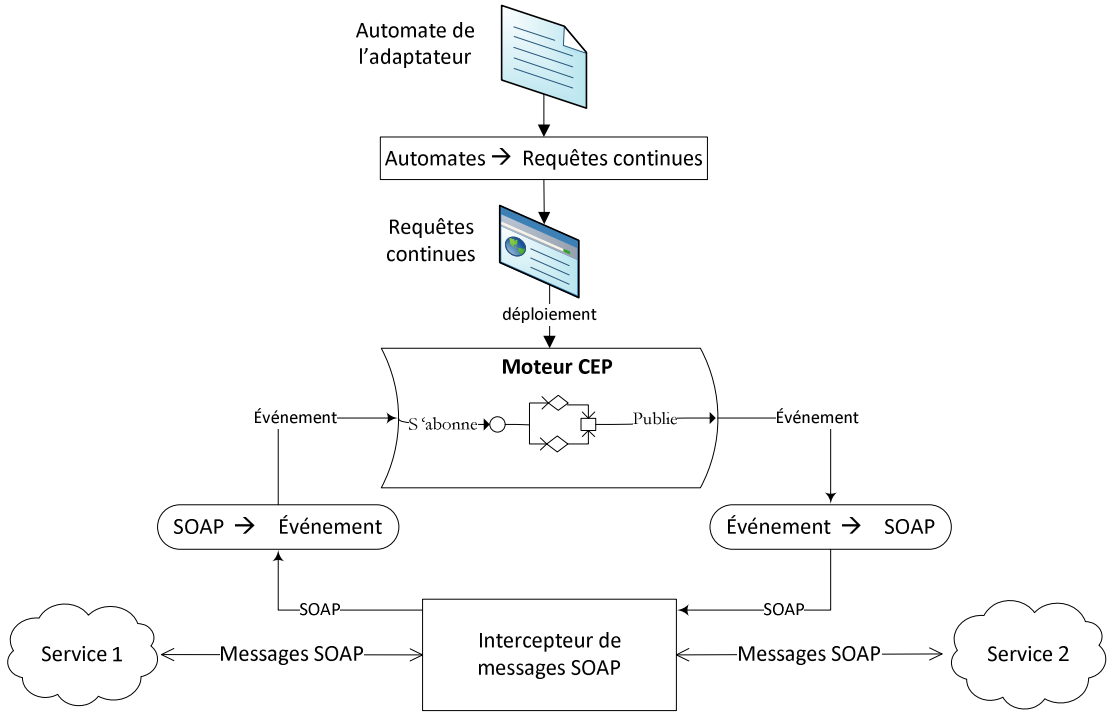

Figure 5.4 Architecture conceptuelle de l'approche

5.3.3 Traduction CEP des operateurs d'adaptation

Dans notre architecture, nous avons mis en œuvre un module spécifique pour la transformation d'automates en des requêtes continues. En effet, pour chaque opérateur d'adaptation, nous avons spécifié une méthode qui prend en entrée l'automate qui le décrit, et génère automatiquement une requête continue correspondante. Une fois déployées dans l'engin, l'ensemble des requêtes générées réalisent le fonctionnement et le comportement de l'automate qui spécifie l'adaptateur.

Dans ce qui suit, nous décrivons pour chaque opérateur d'adaptation, la méthode de transformation correspondante d'automates en des requêtes continues.

Opérateur de mise en correspondance

La figure 5.5 fournit une illustration visuelle du principe de translation de l'operateur de

mise en correspondance, d'automates en requête continues.

D'après l'automate qui le spécifie, le procédé de l'opérateur de mise en correspondance consiste à consommer un message provenant d'un service émetteur via l'action ?(m), adapter sa structure par la fonction de transformation F_{match}, puis transmettre le message ainsi obtenu au service destinataire par le biais de l'action !(m').

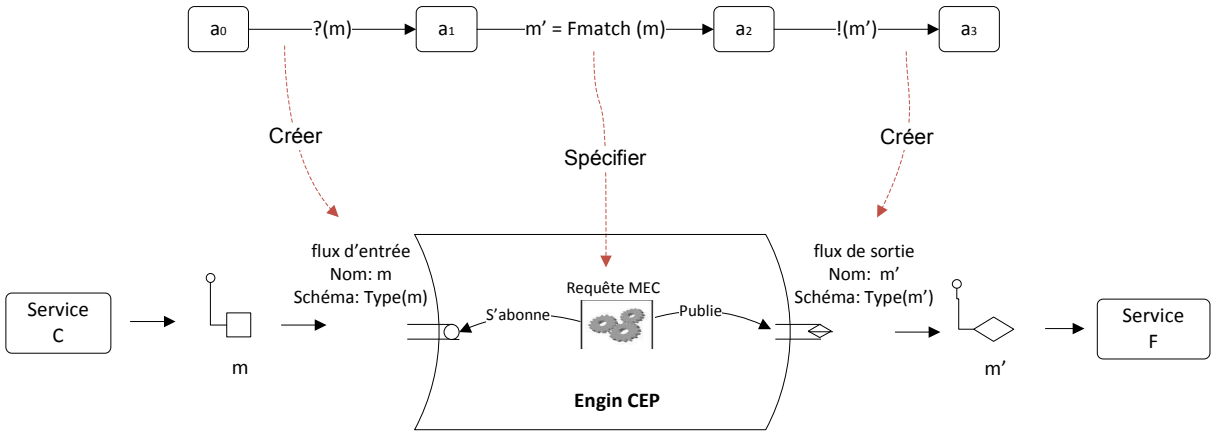

Figure 5.5 Traduction CEP de l'operateur MEC

Pour traduire ce processus en requête continue, l'idée consiste à procéder comme suit :

- Créer un flux d'entrée qui correspond à l'action ?(m) . La correspondance signifie que le flux créé est du même nom et de même schéma de données que celui du message associé à l'action en question. Ce flux est utilisé pour faire introduire dans le moteur CEP les messages de type m, provenant du service émetteur,

- Créer un flux de sortie qui correspond à l'action !(m'). Ce flux sert à transmettre au service destinataire le message produit par moteur,

- Spécifier une requête continue qui s'abonne au flux d'entrée étiqueté par m, et qui attend l'arrivée des messages. Dès qu'un message est disponible sur le flux, la requête le consomme, puis transforme sa structure par le biais de la fonction de transformation F_{match}. Le résultat de la transformation est publié dans le flux de sortie étiqueté par m'.

La figure 5.6 illustre le flux du message Devis depuis son arrivée dans le flux d'entrée Devis_IN jusqu'à sa publication dans le flux de sortie Devis_Out, en passant à travers la requête de mise en correspondance (Requête MEC). Celle-ci est codée par le langage de traitement continue Coral8 [CCL08].

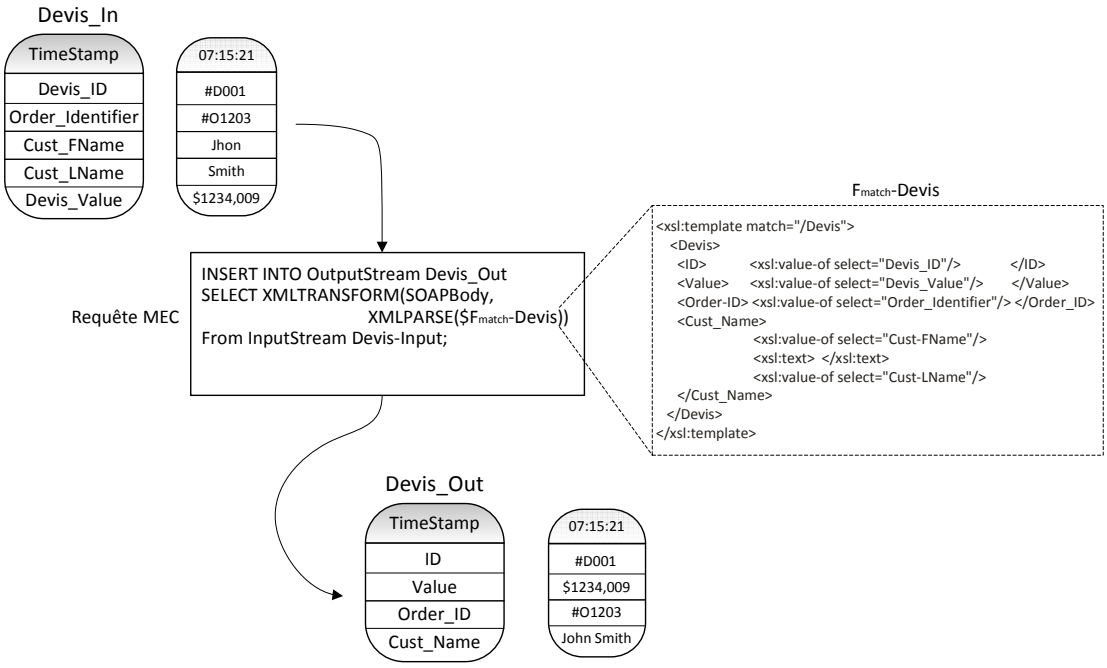

Figure 5.6 Flux de messages devis à travers la requête MEC

Dans la Requête MEC, la clause INSERT INTO, qui est toujours le premier élément d'une requête continue, indique la destination de sortie de la requête. Dans cet exemple, la destination de données est le flux de sortie Devis_Out.

Toute déclaration de requête doit avoir une clause SELECT, immédiatement après la clause INSERT INTO. La clause SELECT définit le contenu de la sortie de la requête. La Requête MEC utilise une syntaxe spéciale SELECT XMLTRANSFORM…, qui indique la transformation du corps SOAPBODY du message, par la fonction de transformation $F_{Match\text{-}Devis}$ codée en XSLT.

La clause FROM suit la clause SELECT; elle spécifie la source de données à laquelle la requête est abonnée. Dans cet exemple, la source de données est le flux d'entrée Devis_IN.

Opérateur de fraction

La figure 5.7 fournit une illustration visuelle du principe de traduction en *CEP* de l'operateur de fraction.

Comme l'illustre l'automate de la figure 5.7, l'operateur de fraction consiste à consommer un message provenant d'un service émetteur via l'action ?(m), fractionner le message reçu en un nombre de sous-messages par le biais des fonctions de transformation $F_{Extract1}$ et $F_{Extract2}$, puis transmettre les messages ainsi obtenus à leurs destinataires via les actions $!(m_1)$ et $!(m_2)$.

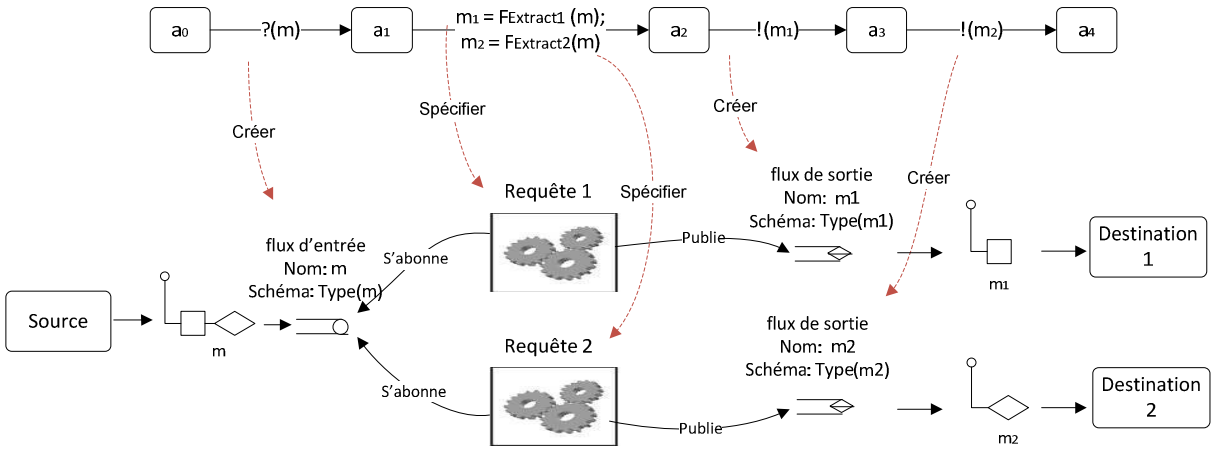

Figure 5.7 Traduction CEP de l'opérateur de fraction

En *CEP*, cette procédure est traduite comme suit :

- Créer un flux d'entrée qui correspond à l'action ?(m),
- Pour chaque action d'envoi de message, créer un flux de sortie correspondant,
- Pour chaque flux de sortie créé, spécifier une requête qui s'abonne au flux d'entrée et qui publie dans le flux de sortie associé. Dès qu'un message est disponible sur le flux d'entrée, les requêtes abonnées à ce flux le consomment. Chacune de ces requêtes extrait une partie du message qui la concerne et publie le résultat dans le flux de sortie à laquelle elle est associée.

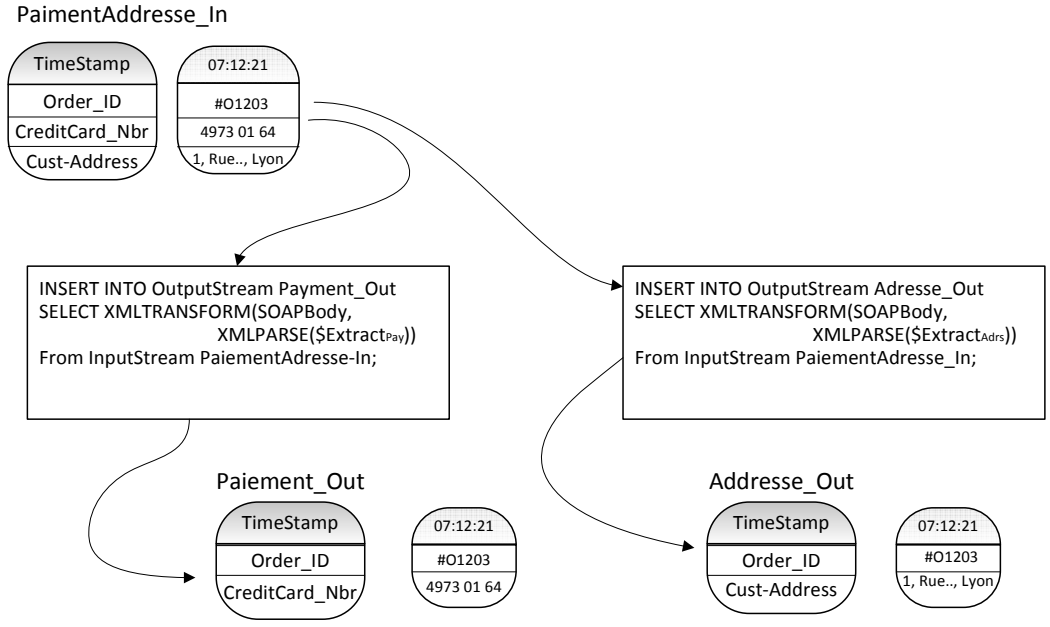

Figure 5.8 Flux du message Paiement-Adresse à travers les requêtes de fraction

La figure 5.8 illustre le fonctionnement de l'opérateur de fraction sur le message $Paiement - Adresse$ arrivé à travers le flux d'entrée PaiementAddresse_IN. Au moment de son arrivée, le message est consommé par les deux requêtes qui réalisent l'operateur de fraction. La première requête extrait les détails de paiement à partir du message consommé, et publie le résultat dans le flux Paiement_Out. Tandis que la deuxième requête extrait l'adresse de facturation à partir du message et publie le résultat dans le flux de sortie Addresse_Out.

Opérateur de fusion

La figure 5.9 fournit une illustration visuelle du principe de la traduction en requête continues de l'operateur de fusion.

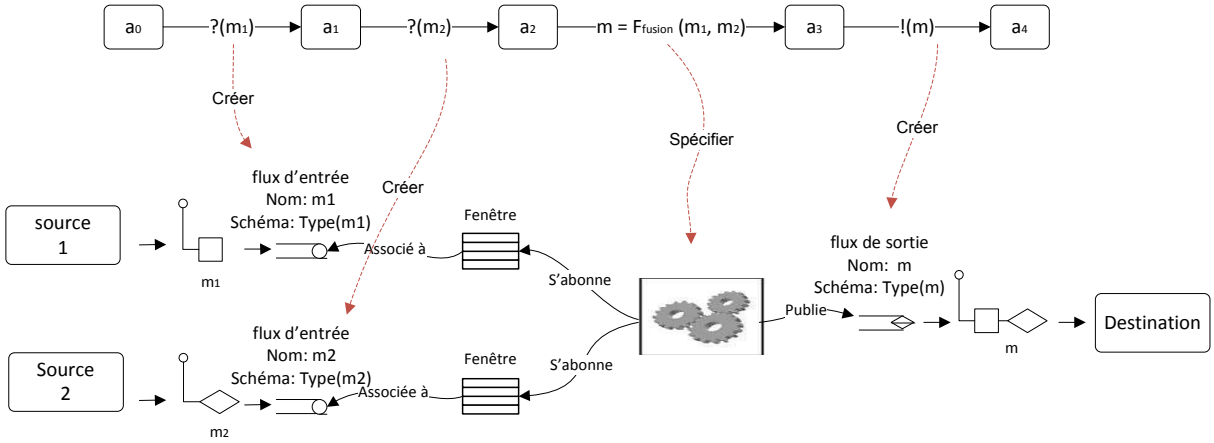

Figure 5.9 Traduction CEP de l'operateur de Fusion

L'opérateur de fusion consiste à consommer n messages provenant de n sources de données différentes, puis les fusionner en un seul message. Etant que les messages sont prévenus des différentes sources de données, à différents temps, plusieurs sessions de conversation, dont chacune est initiée par un utilisateur, peuvent être tournées en même temps entre le consommateur et le fournisseur. Un *critère de corrélation* est introduit dans l'opérateur de fusion afin d'identifier appartement à la même session de conversation lors de la fusion des messages. Ce critère est spécifié par le concepteur de l'adaptateur. Dans le cas des achats en ligne, le critère de corrélation est, la plupart des fois, le numéro de commande.

En CEP, l'idée consiste à créer un flux d'entrée ainsi qu'une fenêtre pour chacune de ces sources. Un message provenant à travers un flux d'entrée est stocké dans la fenêtre associée à ce flux. La requête de fusion est abonnée aux fenêtres dans lesquelles les messages arrivant sont stockés. La requête vérifie en continu la disponibilité de n messages appartenant à des fenêtres différentes les unes des autres, et vérifiant le critère de corrélation spécifié dans

l'opérateur. Une fois ce critère vérifié sur n messages, la requête assemble ces messages en un seul, et le publie dans le flux de sortie associé. Ce dernier sert à son tour à transporter le message ainsi obtenu à son destinataire.

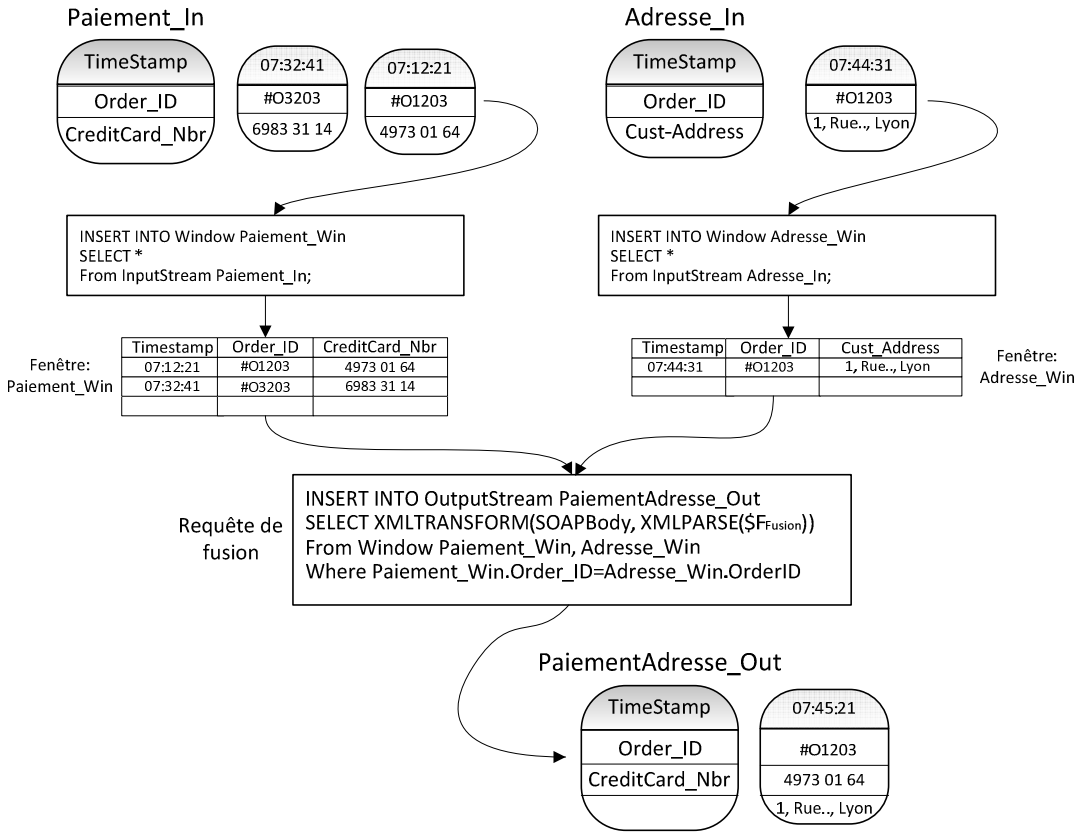

Figure 5.10 Flux des messages Paiement et Adresse à travers la requêtes de fusion

La figure 5.10 illustre le flux de messages à travers l'operateur de fusion. Le flux d'entrée Paiement_In sert à introduire dans le moteur CEP les messages qui concernent le paiement d'une commande, alors que le flux d'entrée Adresse_In introduit dans le moteur les messages qui concernent l'adresse de facturation. Les messages arrivant à travers le premier flux (respectivement le deuxième flux) sont stockés dans la fenêtre Paiement_Win (respectivement dans Adresse_Win). Selon la figure 5.10, à un moment donné, deux messages appartement à deux commandes différentes (#03203 et #1203) sont arrivés à travers le premier flux et stockés dans la fenêtre Paiement_In. Ainsi un message appartenant à la commande #01203 est arrivé à travers le flux d'entrée Adresse_In et est stocké dans la fenêtre Adresse_Win. Etant donné que dans cet exemple, nous nous spécifions le numéro de commande comme critère de corrélation, la requête de fusion assemble les deux messages stockés successivement dans la première et la deuxième fenêtre, et ayant le même numéro de commande. Ensuite, le message obtenu est publié dans le flux de sortie

PaiementAdresse_Out. Une fois fusionnés, les deux messages sont éliminés respectivement des deux fenêtres.

Opérateur d'agrégation

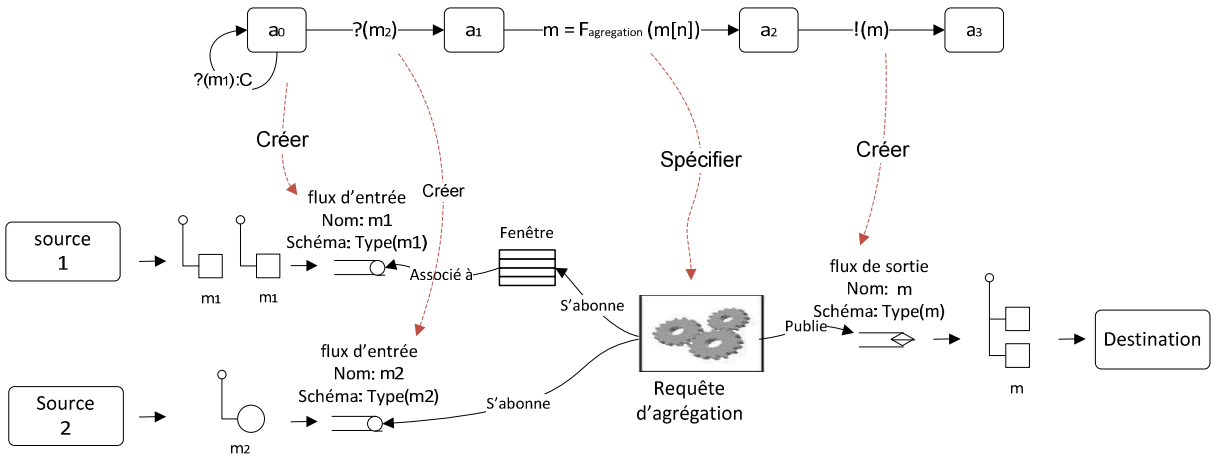

Figure 5.11 Traduction CEP de l'opérateur d'agrégation

La figure 5.11 illustre le processus de transformation traduction en requêtes continues l'operateur d'agrégation. Il s'agit de :

- Créer un flux d'entrée qui correspond à l'action $?(m_1) : C$, ainsi qu'une fenêtre pour stocker les messages provenant à travers le flux d'entrée.
- Créer un flux d'entrée qui correspond à l'action $?(m_2) : C$.
- Spécifier la requête d'agrégation : elle s'abonne à la fenêtre dans laquelle les messages consommés par l'action $?(m_1) : C$ sont stockés, et au flux d'entrée qui correspond à l'action $?(m_2) : C$. Dès qu'un message provient sur ce flux, les messages stockés dans la fenêtre, et liés avec le message arrivé par le même critère de corrélation, sont agrégés en un seul message et le résultat est publié dans le flux de sortie associé, qui à son tour transporter le message à son destinataire.

Dans la figure 5.12, les messages arrivant à travers le flux Order_In sont stockés dans la fenêtre Order_Win. A l'arrivée du message signalant la complétude de la commande numéro #03203 à travers le flux Validation_In, les messages stockés dans la fenêtre et portant le même numéro de commande (#03203), qui est considéré dans cet exemple comme critère de corrélation, sont agrégés en un seul message. Ainsi, le message obtenu composé des deux articles : $Item_1$ et $Item_2$, est publié dans le flux Order_Out.

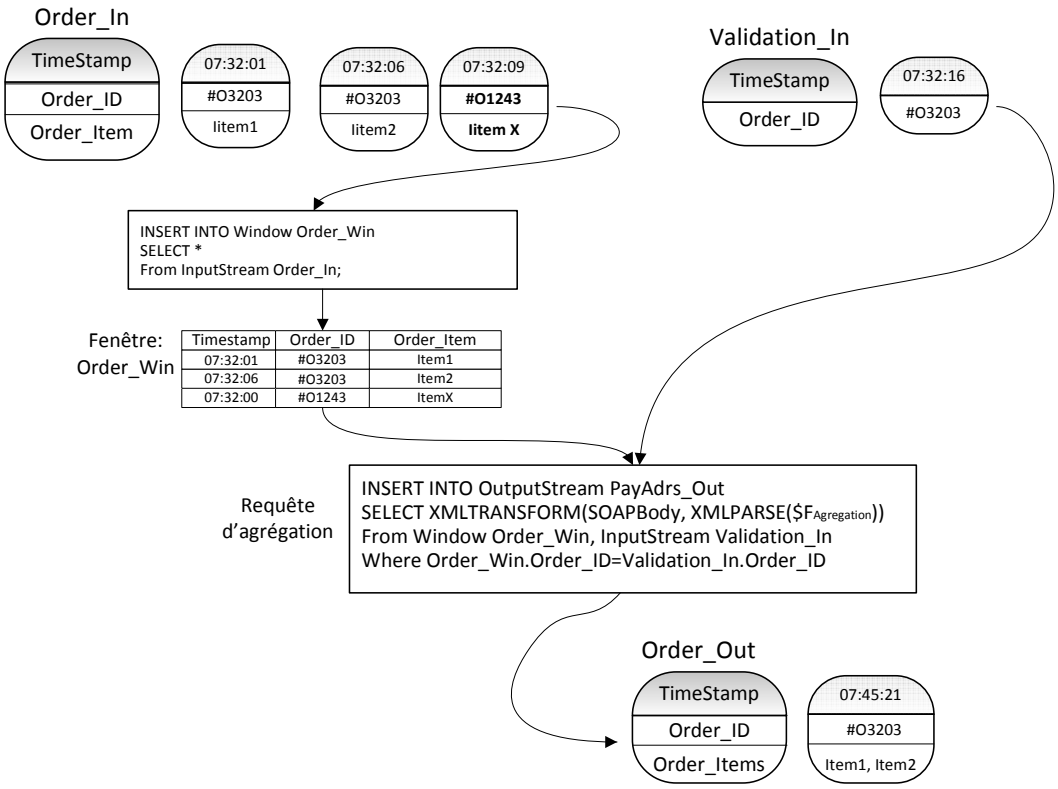

Figure 5.12 Flux de messages à travers l'operateur d'agrégation

5.4 Mise en œuvre du prototype

Nous proposons un canevas, *AdaptTer*, mettant en œuvre notre proposition pour la conception et le développement d'adaptateurs de services.

Comme illustrée auparavant, notre démarche de conception et de développement d'adaptateurs s'appuie sur une formalisation des interfaces à base d'automates. Cette démarche de développement couvre l'ensemble du cycle du développement d'adaptateurs, de la conception au déploiement. Elle s'appuie sur une association des modèles créés, ou PIM (*Plateform Independant Model*), qui modélisent l'aspect fonctionnel et comportemental d'un adaptateur, avec des modèles PM (*Plateform Model*), transformés pour obtenir enfin un modèle d'application spécifique, dit PSM (*Plateform Specific Model*). Des outils de génération automatique de code permettent de générer le code de l'adaptateur directement à partir de ces modèles.

En résumé, notre démarche pour la mise en œuvre d'adaptateurs est entièrement fondée sur les modèles et leurs transformations. Elle se déroule en deux grandes étapes :

- **Conception**. Consiste à réaliser un modèle d'adaptateur indépendant de toute plate-forme

(PIM), et exprimé en automate. Cette réalisation est obtenue comme résultat du processus de détection des incompatibilités et de génération d'adaptateurs (voir le chapitre 4).

- **Transformation**. Consiste à projeter le modèle de l'adaptateur obtenu en termes d'automate, suffisamment détaillé, vers un modèle spécifique (PSM). Les caractéristiques de fonctionnement et de comportement qui ont été spécifiées de façon générique sont alors converties pour tenir compte des spécificités de la plateforme cible. Dans notre cas une plate-forme mettant en œuvre technologie CEP.

La génération du code, s'effectue à l'aide de composants cartouches *(Templates)*. Chaque cartouche étant conçue pour générer des requêtes continues.

La section 5.4.1 décrit l'architecture logicielle du prototype qui comporte deux environnements : un environnement de génération et un environnement de déploiement. La section 5.4.2, présente les détails d'implantation du canevas, et la section 5.4.3 présente une démonstration.

5.4.1 Architecture logicielle

La figure 5.13 schématise l'architecture du canevas proposé. Etant données deux services respectivement consommateur A et fournisseur B dont les interfaces incompatibles, le canevas régit le processus du développement d'un adaptateur entre A et B, depuis sa conception jusqu'à sa mise en œuvre.

Le canevas comporte deux environnements différents : un environnement de conception *(partie basse de la figure)*, et un environnement de déploiement *(partie haute de la figure)*.

L'environnement de conception d'adaptateurs comprend les modules ci-après :

- Le Détecteur des incompatibilités. permet de comparer deux automates décrivant respectivement un service consommateur et un autre fournisseur afin d'identifier leurs incompatibilités. La détection des incompatibilités s'appuie sur les *Expressions des incompatibilités* stockées dans une base de données.

- Le Générateur des adaptateurs. s'appuie sur le résultat de la détection des incompatibilités pour générer automatiquement, le modèle d'automate qui spécifie le fonctionnement et le comportement de l'adaptateur. Cette génération se fait par la configuration et la composition de l'ensemble des operateurs d'adaptation associés aux incompatibilités détectées.

- Le Générateur des requêtes continues. s'appuie sur le modèle exprimé en automates de l'adaptateur. Il traduit ce modèle en requêtes continues à l'aide de cartouches de code. Les requêtes générées représentent le code exécutable de l'adaptateur concret.

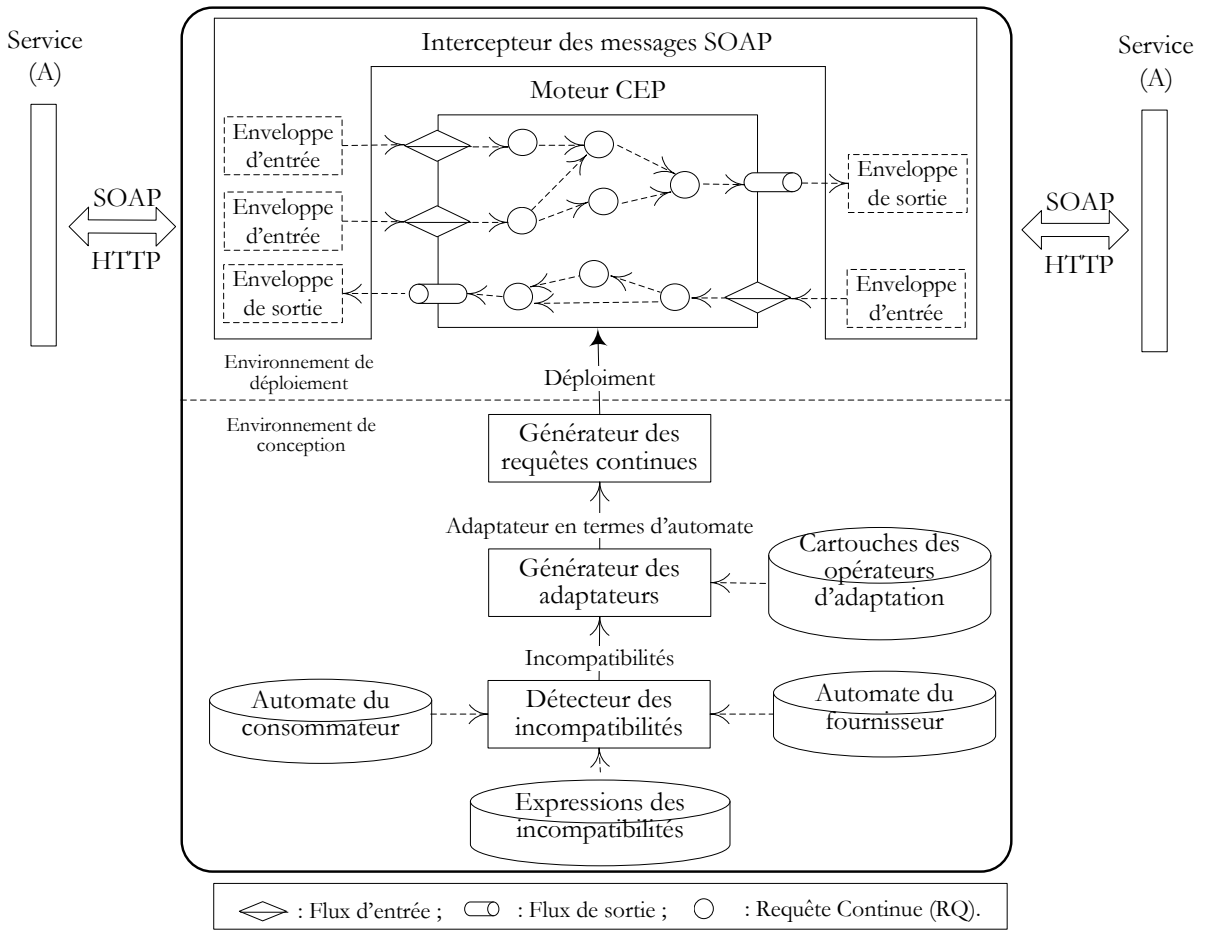

Figure 5.13 Architecture logicielle du canevas AdaptTer

L'environnement de déploiement comprend le Moteur CEP et l'Intercepteur de messages SOAP:

- Le Moteur CEP héberge et exécute les requêtes continues générées. Il est l'élément principal dans l'environnement de déploiement. Il fournit le service de réception, de corrélation, d'analyse et de traitement des messages en fonction de requêtes déployées.

- Alors que les services Web interagissent les uns avec les autres par le biais de messages SOAP, l'adaptateur intermédiaire doit aussi être capable de consommer, d'analyser et de transmettre des messages SOAP. Pour ce faire, nous avons mis en place l'Intercepteur des messages SOAP. Celui-ci est chargé de capter les messages provenant du service émetteur, puis de les acheminer sur les flux d'entrée correspondants dans le moteur CEP.

Ainsi, il est responsable de récupérer les messages produits par le moteur CEP à travers les flux de sortie afin de les transmettre au service destinataire.

5.4.2 Détails d'implantation

Notre canevas contient 14000 lignes de codes *Java* pour 72 classes reparties dans 5 paquetages *Java*. La figure 5.14 illustre les différents paquetages du canevas ainsi que les dépendances entre ceux-ci. Le paquetage AdaptTer est le paquetage principal du canevas. Il contient les classes qui implantent les composantes graphiques utilisés.

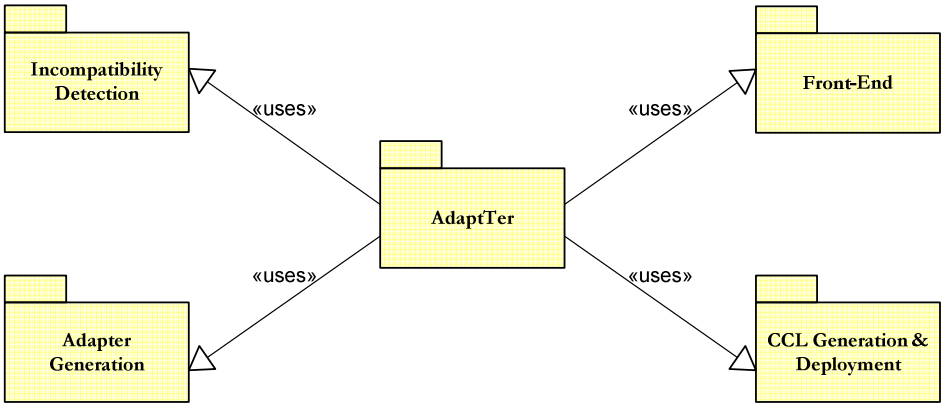

Figure 5.14 Digrammes de paquetages du canevas AdaptTer

Le paquetage Front-End contient les classes qui implantent l'interface Homme-machine du canevas. Ceci permet la lecture d'automates fournis par un utilisateur, ainsi que leur affichage dans un éditeur graphique, la visualisation ainsi que l'export en XML des résultats de la détection des incompatibilités et de la génération des adaptateurs.

Le paquetage Incompatibility Detection contient les classes qui implantent l'algorithme de détection des incompatibilités entre deux automates. Alors que, le paquetage Adapter Generation comprend les classes qui permettent la génération de l'adaptateur sous forme d'un automate.

Le paquetage CCL Generation & Deployment comprend des classes qui permettent la transformation en requêtes continues d'un automate spécifiant un adaptateur. D'autres classes permettent de contrôler le moteur CEP utilisé (*Coral8* [CCL08]). Cela comprend, le démarrage du moteur, ainsi que le déploiement du module de requêtes générées.

Nous détaillons ci-dessous la manière dont les automates sont représentés.

Représentation des automates

La figure 5.15 montre la représentation UML du modèle de l'automate adapté à notre définition des automates.

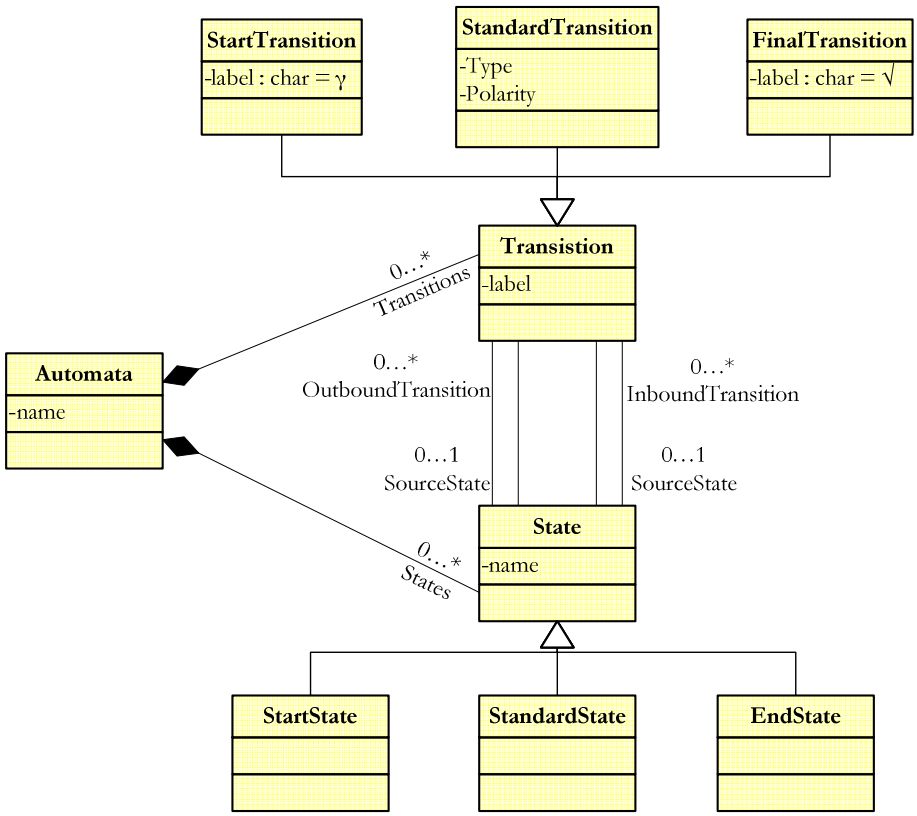

Figure 5.15 Modèle UML des automates

Un automate est modélisé par la classe Automata, où l'attribut *name* est l'attribut qui représente le nom de l'automate. Un automate est composé de plusieurs états (0…* States) et transitions (0…* Transitions), représentés respectivement par les classes State et Transition.

La classe transition est la généralisation des trois classes suivantes : StartTransition, FinalTransition, et StandardTransition. Une transition de la classe StartTransition (respectivement de FinalTransition) est caractérisée par une étiquette de valeur constante γ (respectivement $\surd$). Une transition de la classe StandardTransition est caractérisée par plusieurs attributs, notamment, l'étiquette du message associé (*label*), le type structurel du message *Type*, et sa polarité *Polarity*.

Un état est caractérisé par un attribut *label*. Les états sont classés en StartState, FinalState, et StandardState, selon qu'ils sont initial, final, ou intermédiaire.

Une transition ne peut avoir plus d'un état source (*0…1 SourceState*) (respectivement destinataire (*0…1 TargetState*)). Cependant un état peut avoir plusieurs transitions sortantes (*0…* OutboundTransistion*), et plusieurs transitions entrantes (*0…* InboundTransistion*).

En partant de ce modèle, nous avons généré (à l'aide de l'outil *Topcased*[15]) le schéma XML correspondant. Ce schéma définit la représentation sur laquelle un utilisateur s'appuie pour formater son automate afin de l'importer dans le canevas. Tout automate dont le document XML n'est pas conforme à ce schéma, est rejeté par le canevas.

Support visuel

Le canevas fournit un support visuel permettant de contrôler graphiquement les différentes étapes du développement des adaptateurs. Notamment, le canevas offre les fonctionnalités suivantes:

- Transformer les représentations XML des automates en des représentations graphiques,
- Visualiser les incompatibilités détectées entre deux automates,
- Afficher le modèle de l'adaptateur généré.

Pour réaliser ces fonctionnalités, nous nous sommes appuyés sur la plate-forme GMF[16] (*Graphical Modeling Framework*) basée sur EMF[17] (*Eclipse Modeling Framework*) et GEF[18] (*Graphical Editing Framework*).

EMF offre un cadre et un support à la méta-modélisation, qui consiste à spécifier un méta-modèle sous forme d'un modèle EMF (généré à partir d'un modèle UML) et à en produire une implémentation en Java afin de représenter les instances de ce méta-modèle et de les manipuler.

GEF est un environnement pour le développement des éditeurs graphiques. Il permet à un utilisateur de définir une sémantique graphique permettant de faire le lien entre la visualisation et le modèle EMF correspondant. Il autorise ensuite de spécifier une partie du code Java permettant d'implémenter l'interface graphique à partir des classes produites par EMF et GEF.

GMF permet de relier le deux les deux plates-formes EMF et GEF. Une fois la mise en

[15] http://topcased.gforge.enseeiht.fr/
[16] http://www.eclipse.org/modeling/gmf/
[17] http://www.eclipse.org/modeling/emf/
[18] http://www.eclipse.org/gef/

correspondance entre le modèle EMF et l'information graphique associée effectuée, GMF fournit un générateur des modèles graphiques. Ce dernier autorise d'implanter les détails du modèle permettant d'aboutir à la génération finale sous la forme d'un plug-in directement utilisable. Ce dernier contient tous les paramètres et les définitions du modèle à représenter. Une fois créé, il génère un éditeur et permet de lire des fichiers textuels conformes au modèle EMF, et les représenter graphiquement à l'aide des classes GEF qui établissent les liens entre les représentations textuelles d'une part et visuelles d'autre part. Les instances du modèle sont sauvegardées dans une description XMI [19] (*XML Metadata Interchange : est un standard pour l'échange d'informations de métadonnées UML basé sur XML)*.

C'est d'après ce principe de développement que nous avons implanté le support visuel du canevas. Nous avons généré d'abord les modèles EMF correspondants aux modèles UML représentant les automates, les incompatibilités, et les adaptateurs, ensuite, pour chaque élément du modèle nous avons défini une sémantique graphique. Par exemple, des formes géométriques pour les états d'un automate, des traits pour les transitions, etc. Enfin, nous avons généré d'après GMF, l'éditeur censé lire et afficher les instances de nos modèles (voir la section 5.4.3).

Plate-forme Coral8

En ce qui concerne l'environnement de déploiement, nous nous sommes servis de l'infrastructure Coral8[20], une infrastructure majeure dans le domaine de traitement d'événements complexes. Coral8 fournit :

- Un moteur CEP pour l'exécution des requêtes continues,
- Une API de fonctions pour la création, la compilation, et le déploiement des requêtes continues à partir d'un projet Java,
- Une interface graphique pour le contrôle d'exécution des requêtes continues.

Dans l'environnement de conception, le module de génération et de déploiement des requêtes continues communique avec l'infrastructure Coral8 par le biais d'une API. Cette API permet de déployer dans le moteur CEP, des requêtes continues générées à partir de l'environnement de conception (voir les figures 5.20 et 5.21).

En détails, les requêtes continues générées sont stockées dans un fichier d'extension ".ccl", appelé module des requêtes. Une fois que le module des requêtes est compilé, un fichier d'extension ".ccx" est généré et déployé dans le moteur.

[19] http://www.omg.org/spec/XMI/2.1.1/
[20] http://www.coral8.com/

Le processus de compilation permet de générer le code exécutable de requêtes ainsi de créer les flux d'entrée, et de sortie, et les fenêtres associés à celles-ci. Pour chaque flux d'entrée/sortie, l'URI correspondant est retourné. Un URI d'un flux d'entrée est utilisé par l'intercepteur de messages SOAP pour publier dans le moteur les messages provenant du service émetteur, alors qu'un URI d'un flux de sortie est utilisé par l'intercepteur pour transporter au destinataire les messages produits par le moteur.

5.4.3 Démonstration

Les fonctionnalités du canevas *AdaptTer* sont offertes via des IHM dont nous donnons des copies d'écran. Celles-ci sont :

- La lecture d'un couple d'automates qui décrivent respectivement un service consommateur et un service fournisseur (voir les figures 5.16 et 5.17)
- La détection des incompatibilités par comparaison d'automates (voir la figure 5.18)
- La génération de l'adaptateur exprimé en automate (voir la figure 5.19)
- La génération du code CCL de l'adaptateur (voir la figure 5.20)
- Le déploiement du code CCL généré dans l'engin CEP (voir la figure 5.21)

Le canevas fonctionne de la manière suivante : l'utilisateur sélectionne le couple d'automates à comparer par le menu « Canevas Tools – Import automata» de l'application qui ouvre une fenêtre de navigation dans le système de fichiers local (voir la figure 5.16). Les formats XML des deux fichiers sélectionnés doivent être conformes au schéma XML d'automates acceptés par l'application. Dans le cas où l'un ou les deux fichiers sont mal formatés, le message d'erreur correspondant s'affiche. Autrement, l'application lit les deux fichiers et génère les représentations visuelles correspondantes (voir la figure 5.17).

Après l'affichage, il s'agit de lancer l'étape de la détection des incompatibilités. Dans la figure 5.18, l'utilisateur clique sur le menu « Canevas Tools – Detect incompatibilities», ce qui lance l'algorithme de détection qui compare les deux automates. La comparaison s'appuie sur les représentations XML des automates. Le résultat de la détection est obtenu sous forme textuelle, et est intégré dans le fichier de l'instance de l'éditeur. Ce qui permet d'afficher ce résultat graphiquement sur l'éditeur. Après la détection des incompatibilités, le concepteur clique sur le menu « Canevas Tools – Generate adapter » pour générer l'adaptateur (voir la figure 5. 19).

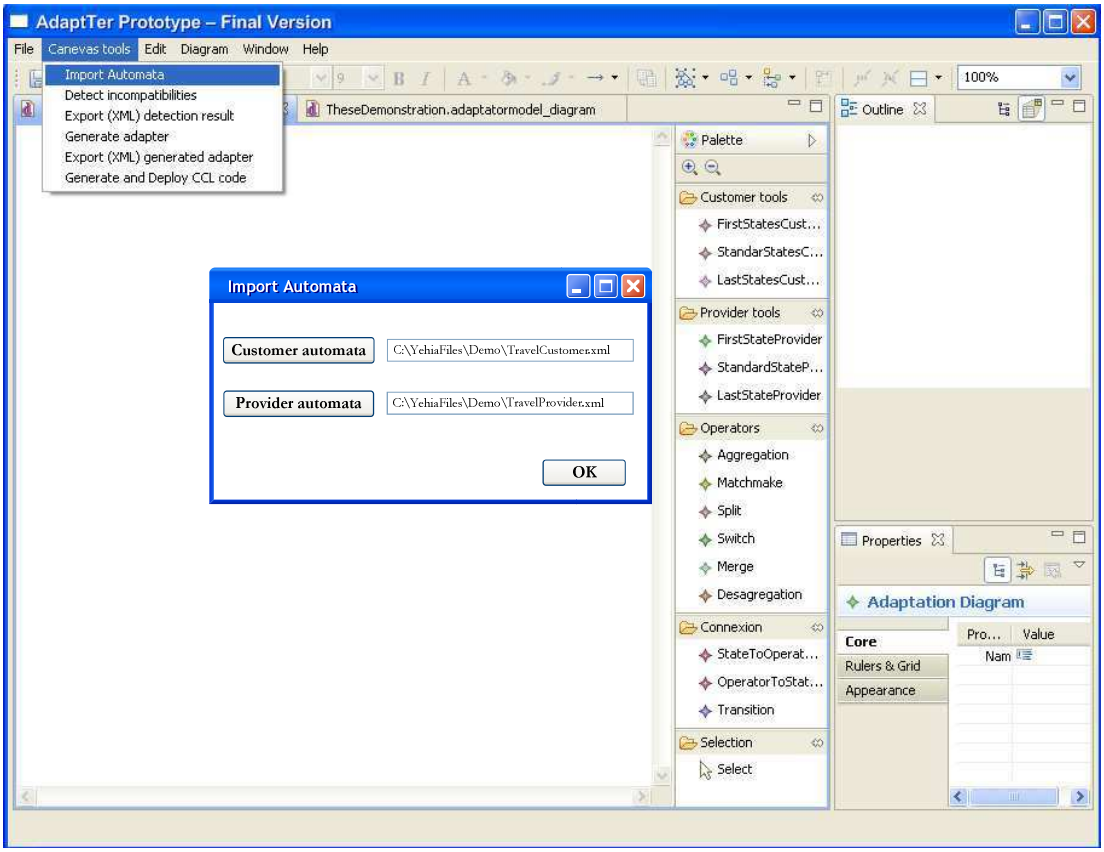

Figure 5.16 Lecture d'automates

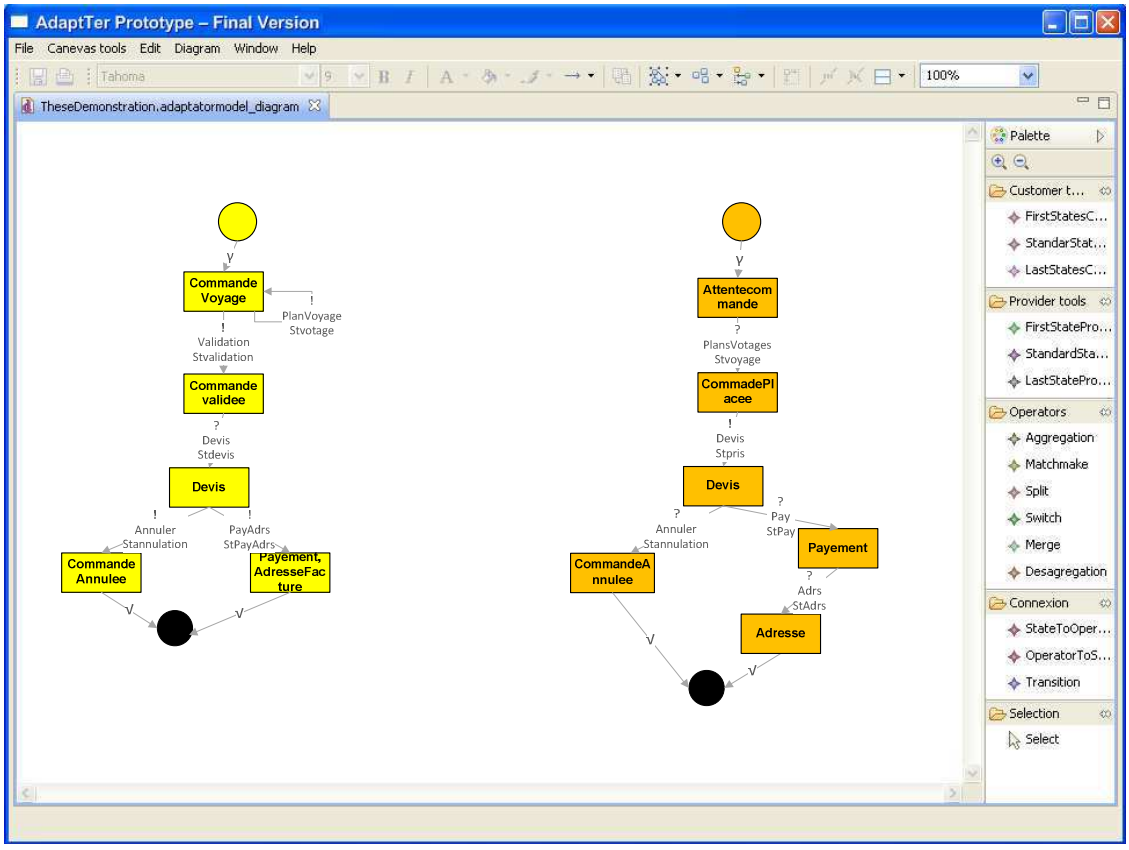

Figure 5.17 Affichage d'automates

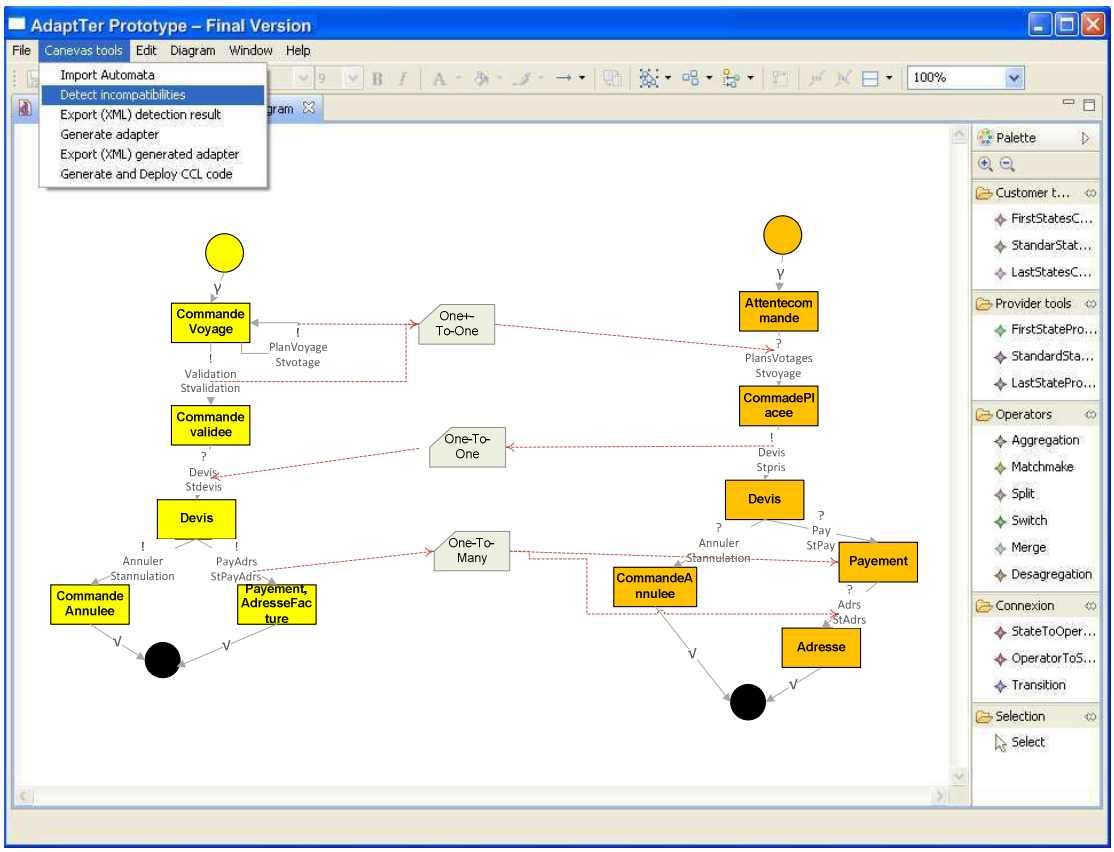

Figure 5.18 Détection des incompatibilités

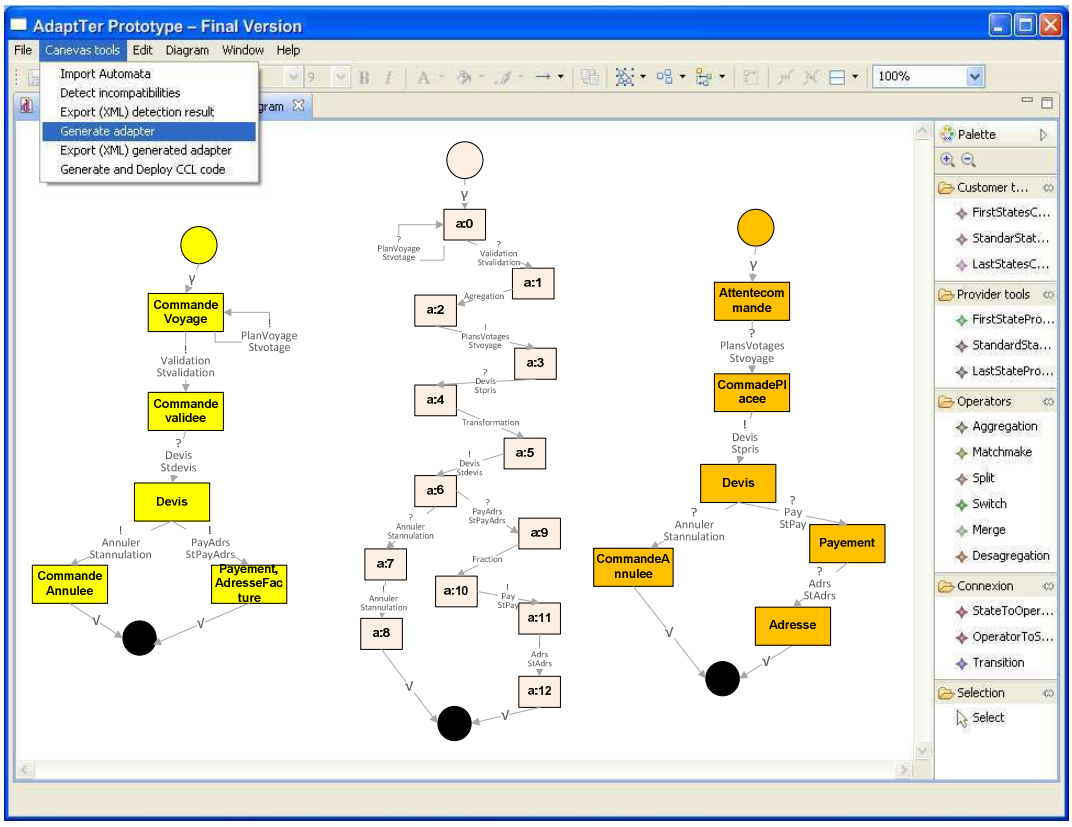

Figure 5.19 Génération de l'adaptateur

Après la génération de l'adaptateur sous forme d'automate, il s'agit de générer et déployer le code exécutable de celui-ci. Pour ce faire, le concepteur clique sur le menu « Canevas Tools – Generate and deploy CCL Code » (voir la figure 5.20). La commande de génération et de déploiement du code CCL comprend implicitement le démarrage du moteur CEP afin de permettre le déploiement du code de l'adaptateur. Elle comprend également une commande du lancement de l'interface graphique (Coral8 studio) fournit par Coral8. Cela permet d'afficher le code généré et de contrôler l'échange des messages passant à travers le moteur (voir la figure 5.21).

Enfin, nous nous somme appuyés sur l'outil SOAPUI[21] pour tester l'adaptateur généré. L'outil permet à partir du WSDL d'un service de générer les messages SOAP par défaut qui permettent d'appeler le service.

Nous avons installé l'outil sur deux machines différentes (voir la figure 5.22) pour simuler respectivement le service fournisseur et le service consommateur. Les messages envoyés/reçus par les deux services passent à travers le moteur CEP. Ainsi, coral8 studio associé au moteur permet de visualiser les messages entrants et sortants dans celui-ci.

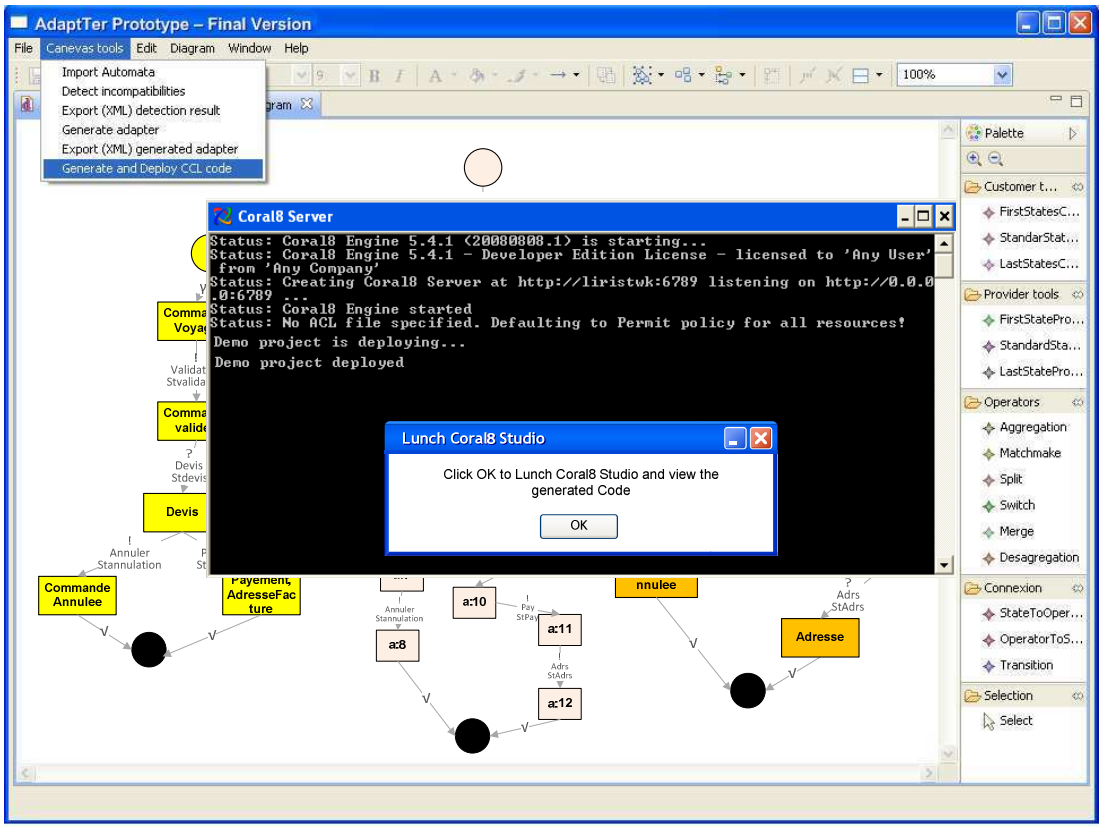

Figure 5.20 Déploiement du code CCL généré

[21] http://www.soapui.org/

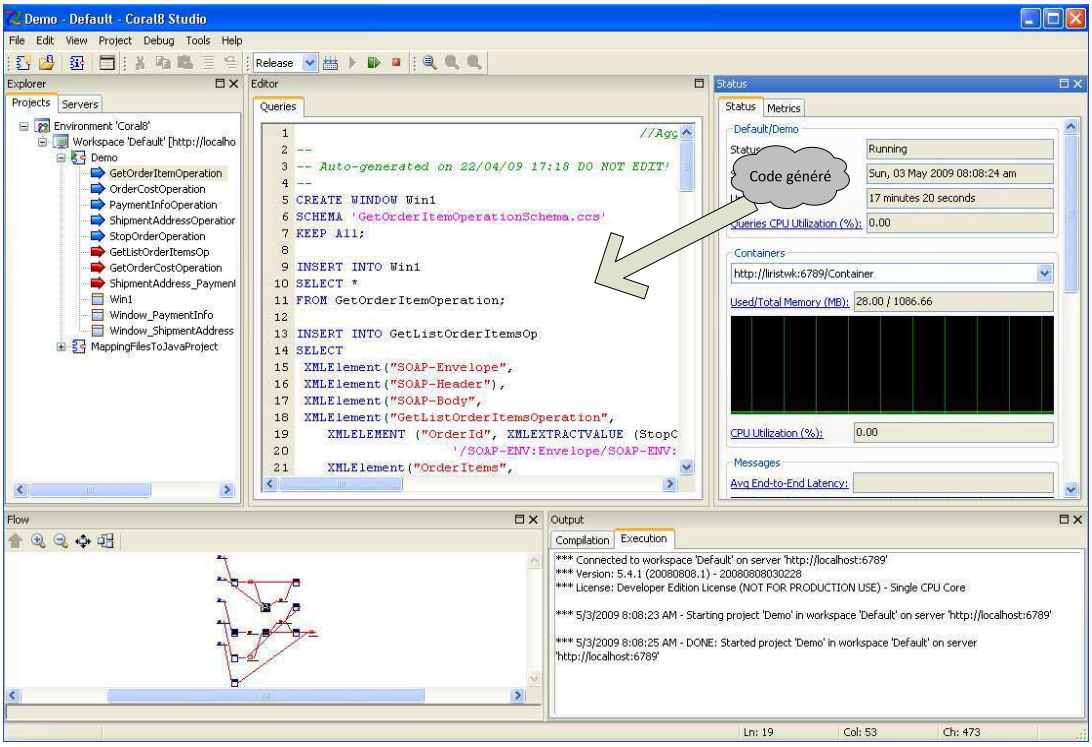

Figure 5.21 Interface graphique du Coral8 studio

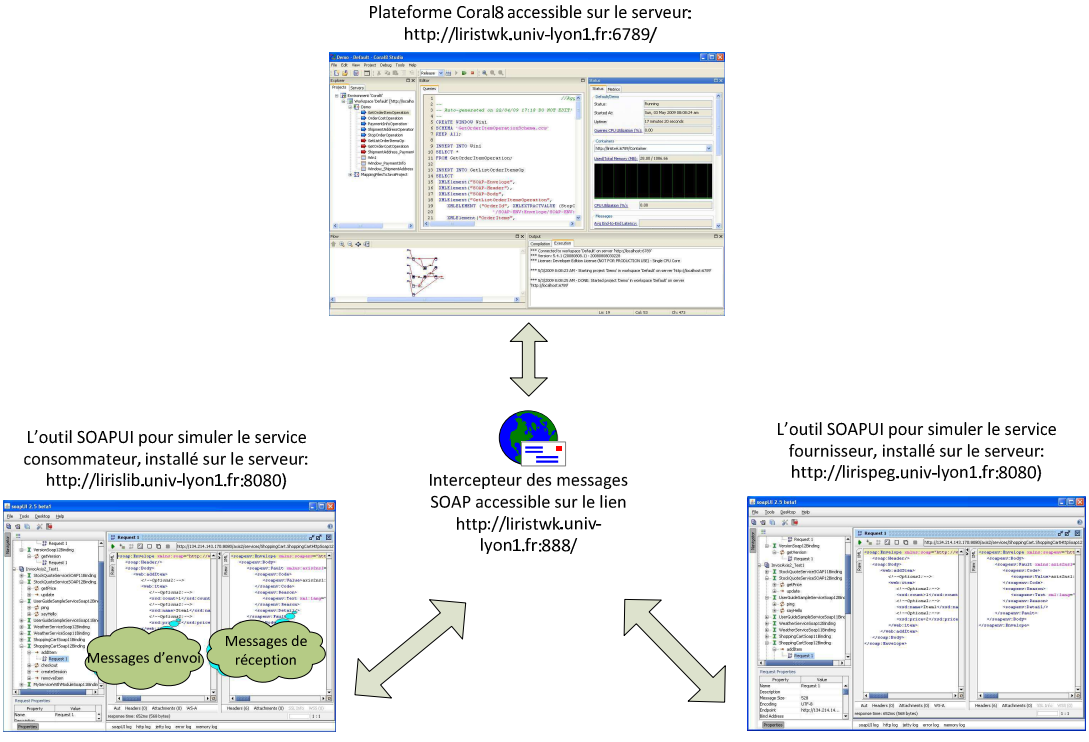

Figure 5.22 Illustration du test d'exécution de l'adaptateur

5.5 Conclusion

Dans ce chapitre, notre contribution est double. En premier lieu, nous avons intégré la technologie CEP avec l'architecture à base de services. Nous avons proposé une méthode pour la mise en œuvre de l'adaptation d'interactions des services dans une infrastructure CEP. En particulier, nous avons fourni une méthode pour la mise en œuvre des opérateurs d'adaptation en termes de requêtes continues.

En deuxième lieu, nous avons mis en œuvre un canevas pour la génération automatique des adaptateurs. Un support visuel a été mis en œuvre pour visualiser les différents étapes de la génération d'un adaptateur depuis la lecture d'interfaces de services jusqu'à le déploiement de l'adaptateur généré.

Le canevas *AdaptTer* a été démontré à plusieurs reprises, notamment lors de la conférence *WWW'08* à Pékin en chine [TFDB08].

Troisième partie

Proposition d’une architecture logicielle multicouche pour faciliter la substitution de services Web

CHAPITRE 6

Architecture logicielle multicouche pour faciliter la substitution de services Web

Sommaire

6.1 Introduction

Dans le chapitre 1 de cette thèse, nous avons motivé notre proposition pour la génération des adaptateurs des services Web par un scénario de substitution de services. Lorsqu'un consommateur remplace son service fournisseur habituel par un autre service, la génération automatique d'un adaptateur entre le service consommateur et le nouveau service permet de décharger le consommateur des détails de l'adaptation. Cependant, la tâche de sélection du nouveau service et la liaison à celui-ci reste à la charge du concepteur du service consommateur.

Dans ce chapitre, nous proposons une architecture multicouche qui s'appuie sur un niveau d'abstraction intermédiaire entre les consommateurs et les fournisseurs afin de faciliter la tâche de substitution [TBFM06, BMT+07].

En particulier, nous proposons d'étudier la notion de communauté faisant référence à un

ensemble de services de même fonctionnalité. Une communauté offre à ses utilisateurs (développeurs d'application, fournisseurs de services, consommateurs) des fonctions pour connaître des informations liées à la communauté, pour sélectionner, puis utiliser un ou plusieurs des services qui y sont enregistrés. Les contributions de ce travail sont :

- Un modèle de communauté formalisé par le biais d'un ensemble de types abstraits. Les opérations associées permettent en particulier d'effectuer, au sein d'une communauté, la sélection du service répondant le mieux à un ensemble de critères de qualité (réputation, fiabilité, etc.).

- Une architecture logicielle s'appuyant d'une part, sur des communautés implantées comme des services, et d'autre part sur un progiciel nommé *OSC (Open Service Connectivity)*. Ce composant peut être vu comme une bibliothèque de fonctions permettant de gérer les interactions entre les consommateurs et les services fournisseurs à travers le modèle de communauté.

Le présent chapitre est organisé comme suit. La section 6.2 présente les motivations du travail rapporté ici et les problèmes posés par l'approche d'*UDDI*[22]. La section 6.3 décrit, de manière générale, une communauté de services web ; une architecture logicielle est ensuite détaillée et son implantation est présentée. Dans la section 6.4, nous proposons une formalisation de la notion de communauté. Dans la section 6.5, nous donnons un aperçu des travaux se rapportant à la notion de communauté, et nous les positionnons par rapport à notre approche. Finalement, la section 6.6 présente les conclusions de ce travail et décrit ses extensions possibles.

6.2 Motivations

Nous présentons tout d'abord dans la section 6.2.1 un scénario illustrant le problème posé par la substitution, à l'exécution, d'un service par un autre dont les caractéristiques non fonctionnelles correspondent mieux aux besoins de l'application cliente. La section 6.2.2 discute du standard *UDDI* et de ses limites.

6.2.1 Scénario

Dans cette section, nous présentons un scénario dont l'objectif est d'illustrer et de motiver notre approche pour la substitution et la sélection de services web. Ce scénario concerne l'organisation d'une compétition sportive. Chaque athlète est muni d'une montre bracelet qui

[22] Universal Description, Discovery and Integration [OAS07]

a deux rôles : (1) avertir le centre de surveillance en cas de défaillance, (2) transmettre les coordonnées (i.e., longitude et latitude) de la localisation de l'athlète en difficulté. Lors de la réception d'un signal de défaillance, le centre de surveillance achemine une ambulance auprès de l'athlète après avoir transformé les coordonnées en une adresse de surface connue de leurs chauffeurs.

Le centre de surveillance met en place une application cliente nommée Urgence qui utilise deux services web Localisation et Ambulances38. Le service Localisation fournit l'opération Coordonnées2Adresse (real, real) : string qui admet en paramètres la longitude et la latitude d'un point et retourne l'adresse correspondante. Le service Ambulance38 dispose de l'opération EnvoiEquipe (string) : string qui admet une adresse en paramètre et qui retourne un message de confirmation de l'intervention de l'équipe médicale auprès de l'athlète en difficulté.

Pendant une exécution de l'application cliente Urgence, le service Ambulance38 ne satisfait plus les critères de qualité attendus (e.g. la fiabilité, les performances, etc.) et pour cette raison doit être remplacé. Pour réaliser cette substitution, le client soumet une requête de sélection à la communauté Secours qui rassemble un ensemble de services dédiés à l'assistance et au transport des personnes blessées. Cette sélection retourne le service qui, parmi ceux de la communauté répond le mieux aux critères de qualité requis par le client (donnés en paramètres de la sélection). Les questions soulevées par ce scénario sont les suivantes :

- Quel jeu d'opérations la communauté expose t-elle ? Pour les fournisseurs de services qui désirent participer ? Pour les clients qui désirent sélectionner des services ?
- Pour chaque communauté, quel est son modèle de qualité ? Quelle est la relation entre ce dernier et celui des services qu'elle rassemble ?
- Quel mécanisme est proposé pour faire que l'application cliente n'ait pas à être modifiée parce qu'un service qu'elle utilise a été substitué par un autre ?

L'objectif de ce chapitre est de répondre à ces questions.

6.2.2 Limites d'UDDI

La mise en œuvre des interactions entre des services web nécessite la découverte, la localisation et l'accès à ces services. Les acteurs participant à ces interactions sont : les applications clientes (qui peuvent être des services) qui cherchent à interagir avec d'autres services et les fournisseurs qui exposent les services qu'ils offrent afin de les rendre

disponibles aux clients.

Selon l'approche *UDDI*, les fournisseurs de services publient les interfaces des services qu'ils offrent (selon une description *WSDL*) dans un annuaire. Un client soumet ensuite à cet annuaire des requêtes de recherche de services. L'annuaire répond en envoyant l'adresse (une *URL*) du service correspondant, s'il en existe. Le client utilise ensuite cette *URL* pour l'envoi des messages traduisant l'invocation des opérations offertes par le service.

Un annuaire *UDDI* contient les informations sur les entreprises et les services qu'elles ont développés et publiés. Il est structuré de la manière suivante :

- Les pages blanches recensent des informations sur l'identité des fournisseurs.
- Les pages jaunes comprennent les descriptions au format *WSDL* des services web déployés par les fournisseurs.
- Les pages vertes qui fournissent des informations techniques détaillées sur les services fournis.

Les requêtes qu'il est possible de soumettre auprès d'un annuaire *UDDI* ont une portée limitée aux aspects fonctionnels des services recherchés. Relativement au scénario vu plus haut, il est possible de chercher des services de type ambulance, mais par contre impossible de retrouver ceux les plus fiables.

De plus, étant donnée l'existence de moteurs de recherche sophistiqués aussi bien pour des Intranets qu'au niveau de l'Internet tout entier, et d'autres technologies pour la gestion de répertoires tels que LDAP [HOW95], la valeur ajoutée apportée par *UDDI* est difficile à identifier. Peu d'architectures à base de services s'appuient aujourd'hui sur *UDDI* et le futur de ce standard est incertain[23].

6.3 Description informelle

Dans cette section, nous décrivons, et ce de manière informelle, la notion de communauté (voir section 6.3.1) avant d'en proposer une architecture logicielle (voir section 6.3.2). Dans la section 6.3.3 nous faisons le point sur le prototype de développement.

6.3.1 Principe général

Le principal objectif d'une communauté de services web est d'offrir un cadre à la

[23] UDDI Business Registry tenu par IBM, Microsoft et SAP a été fermé le 12 janvier 2006.

recherche et à la sélection dynamique de services web et ainsi de pallier les déficiences des annuaires UDDI [BDS05, LUO05]. C'est aussi un moyen pour supporter la composition dynamique (d'un grand nombre) de services [FDD05] ou encore de permettre la substitution (à la conception ou à l'exécution) de services [TBFM06]. Contrairement à un annuaire UDDI, une communauté est spécialisée dans un domaine spécifique. Les services d'une communauté diffèrent entre eux sur des propriétés non fonctionnelles comme par exemple, des critères de qualité (disponibilité, fiabilité, sécurité, réputation, propriétés transactionnelles, etc.) [BHC03, GAR06]. Ces critères de qualité sont définis via un modèle de qualité spécifique à une communauté donnée.

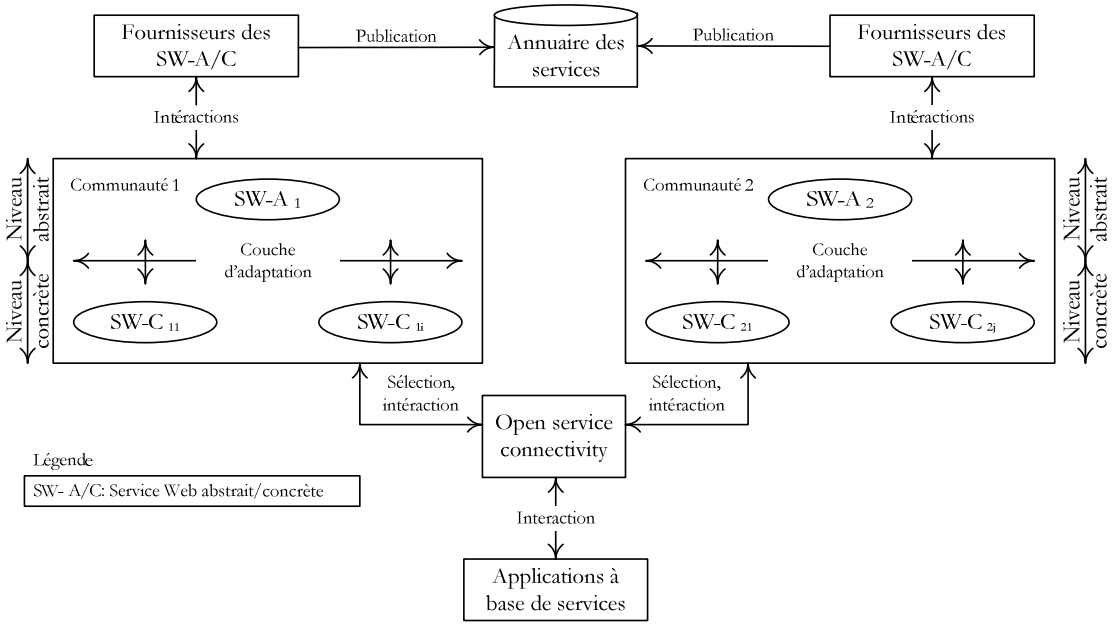

Figure 6.1 Architecture conceptuelle des communautés

Dans notre approche, une communauté de service est perçue comme un moyen pour exposer des descriptions communes d'un caractère fonctionnel désiré sans référer explicitement à un service spécifique. Dans la figure 6.1, un Fournisseur de communauté initialise le regroupement d'un ensemble de services similaires, qualifiés comme concrets, et les exposer par le biais d'une interface qualifiée comme abstraite. Cette dernière offre une vue unifiée de l'ensemble de service regroupés dans la communauté : elle définit d'une manière homogène les interactions dans lesquelles peuvent s'engager tous les services de la communauté. Pour un consommateur, seule l'interface abstraite de la communauté est visible. Ainsi, les interactions entre le consommateur et la communauté sont effectuées en termes de cette interface. Etant donné qu'une interface abstraite n'implémente pas sa fonctionnalité, les requêtes d'un consommateur sont alors déléguées à l'un des services concrets enregistrées auprès de la communauté. Or, les services concrets d'une communauté ne fournissent pas

nécessairement une interface identique à celle du service abstrait de la communauté. C'est pourquoi une couche d'adaptation entre le service abstrait et les interfaces des services concrets a été mise en place. Cette couche consiste en un ensemble des adaptateurs dont chacun est associé à un service concret. Un adaptateur a pour rôle de transformer les interactions provenant d'un consommateur d'une manière adaptée à l'interface abstraite en des interactions adaptées à l'interface du service concret. Cela permet au consommateur d'interagir avec un service concret sans connaître son interface.

L'OSC *(Open Service Connectivity [BMT+07])* est un composant logiciel dont l'objectif étant d'appliquer le principe de pilotes *ODBC* et *JDBC* (utilisés dans des applications de bases de données) dans un environnement orienté service. Plus précisément, il est responsable de gérer les interactions entre les consommateurs d'une communauté et les services de la communauté. Il fournit une API des fonctions – implémentés par la communauté - permettant aux consommateurs la sélection d'un service, son exécution, ainsi sa substitution par un autre d'une façon flexible et transparente.

6.3.2 Architecture logicielle

La figure 6.2 présente l'architecture à laquelle participent une communauté et des fournisseurs de services.

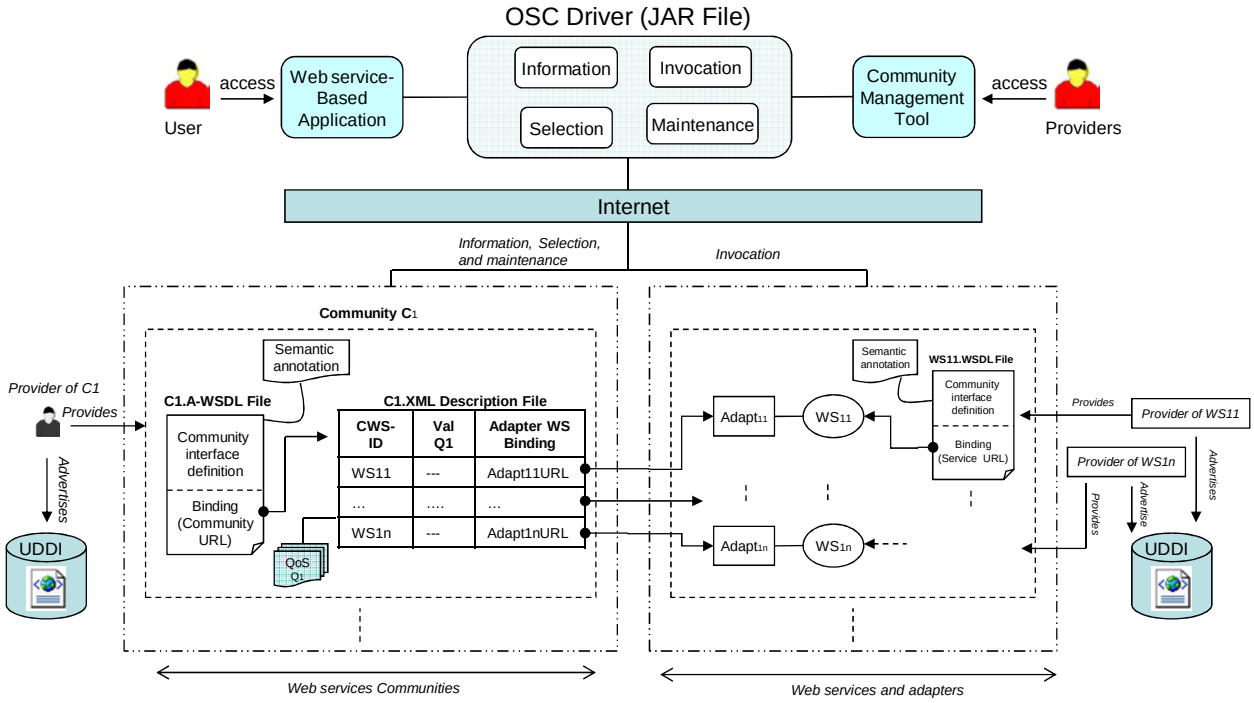

Figure 6.2 Architecture logicielle des communautés

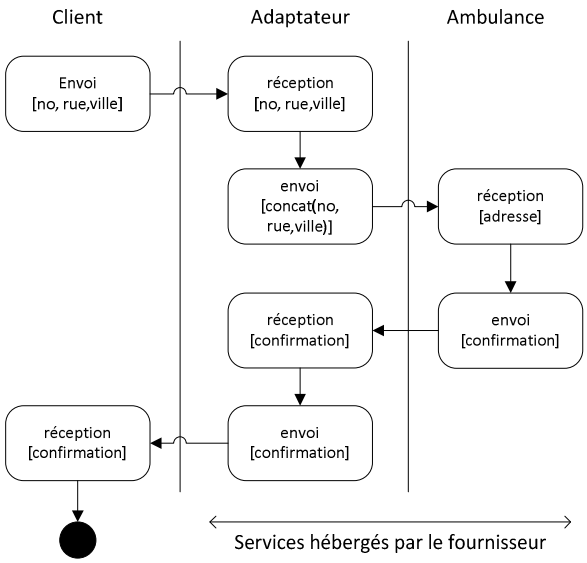

Figure 6.3 Diagramme d'activités décrivant les interactions un client et un fournisseur

La communauté répond aux requêtes de sélection en retournant les adresses URLs des adaptateurs des services sélectionnés. Chaque adaptateur effectue la mise en correspondance entre les opérations de l'interface fournie par le service et celles de l'interface abstraite définie par la communauté. Ainsi, l'exécution du processus d'adaptation est à la charge du fournisseur. Les interactions induites par ce choix d'architecture, entre une application cliente et le service sélectionné par la communauté sont montrées dans la figure 6.3. Les activités décrites correspondent aux envois et aux réceptions de messages. L'interface abstraite de la communauté Secours (voir le scénario, section 6.2.1) décrit ici la prise en charge d'une personne blessée située à une adresse donnée sous forme de trois chaînes de caractères (réception d'un message contenant la rue, le numéro dans la rue et la ville), alors que le service Ambulance38 attend un message contenant une adresse sous forme d'une chaîne de caractères. L'adaptateur, fourni par le service, reçoit le message contenant les trois chaînes de caractères, les concatène avant de transmettre le résultat au service. Ce dernier répond par l'envoi d'un message de confirmation que l'adaptateur transmet, sans modification, au client concerné.

6.3.3 Détails d'implantation

Un prototype a été développé afin de valider la faisabilité de notre approche. Nous avons implanté une application correspondant au scénario illustré dans la section 6.2.1. Nous avons choisi de nous appuyer sur des technologies Java comme outils de développement et de test, d'applications et de services web. Les services web sont déployés en utilisant *AXIS*[24] qui est

[24] http://ws.apache.org/axis/

une implantation Open Source du protocole *SOAP*. Le moteur Axis, qui joue le rôle de client et de serveur *SOAP*, se présente sous forme d'un ensemble de bibliothèques qui implantent l'API *Java JAX-RPC* d'une part, et fournissent d'autre part, des outils pour faire de la journalisation, de la génération et de l'analyse de documents *WSDL*.

Une communauté est implantée selon deux facettes :

- une application web destinée aux fournisseurs de services, c'est-à-dire offrant les opérations pour l'ajout d'un service à la communauté, et pour la gestion des critères de qualité de ce service.

- un service web exposant les opérations destinées aux clients, c'est-à-dire l'opération de sélection d'un service et celle pour l'invocation des opérations du service via son adaptateur qui réalise les mises en correspondance entre l'interface abstraite de la communauté et celle fournie par le service. La partie structurelle de chaque communauté est représentée en XML.

6.4 Modèle de la proposition

Nous introduisons ici un modèle visant la description des communautés de services web, et la définition des opérations offertes. Dans la section 6.4.1 nous proposons une formalisation de la notion de communauté que nous appliquons dans la section 6.4.2 au scénario Urgence.

6.4.1 Formalisation

Dans la suite nous formalisons les types et les opérations associés à la notion de communauté. Nous utilisons un style de formalisation fonctionnel car il fournit un cadre convenable pour décrire les types associés à la notion de service et leurs opérations en faisant abstraction des aspects syntaxiques. Les notations ci-après sont utilisées : $T_1 \longrightarrow T_2$ décrit le type des fonctions dont le domaine est T_1 et le codomaine T_2. $\{T\}$ dénote le type ensemble de T, et $\langle L1 : T1, L2 : T2, \ldots, Ln : Tn \rangle$ désigne le type des n-uplets dont le i^e composant est de type Ti ($1 \leq i \leq n$). Si t est un nuplet de type $\langle L1 : T1, L2 : T2, \ldots, Ln : Tn \rangle$, alors la notation qualifiée $t.L_i$ ($1 \leq i \leq n$) désigne la valeur du composant L_i de t.

Les types abstraits Criterion, Community et Service sont décrits ci-dessous. Le type Criterion modélise la notion de critère de qualité.

type Criterion ⟨
    Name : string, {Le nom du critère, par exemple 'Temps de réponse'.}

valueType : Type {Type est super-type de 'string', 'float', etc.}

{ Le type des valeurs du critère, par exemple 'float'. }

valueRestriction : (Type ⟶ boolean)

{Une restriction éventuelle du type des valeurs, par exemple $\lambda n \bullet n > 0$.}

Unit : ⟨string, string⟩

{Indique, la nature des valeurs du critère et l'unité dans laquelle sont exprimées les valeurs, par exemple⟨'durée', 'millisecondes'⟩.}

⟩

Dans un premier temps, nous avons choisi pour modéliser les critères de qualité associés à une communauté, une approche simple à mettre en œuvre. Une étude doit être menée afin de raffiner ce modèle (en s'appuyant par exemple sur [GAR06]).

type Community ⟨

Name : string {Nom de la communauté}

QualityModel : {Criterion}

{Le modèle de qualité d'une communauté, décrit par un ensemble de critères.}

AbstractInterface : Description-WSDL

{L'interface abstraite de la communauté, il s'agit d'une description WSDL et WSCI que nous ne détaillons pas ici.}

Services : {Service}

{Services est l'ensemble des services enregistrés dans la communauté.}

⟩

Il existe trois modes d'évaluation pour la fonction Value selon que la valeur du critère est stockée par la communauté (la valeur a été donnée par le fournisseur au moment de l'enregistrement du service), ou calculée au moment de la sélection. Dans ce dernier cas, le calcul est basé sur des informations gérées par la communauté (journaux, traces, etc.) ou bien issu de l'invocation d'une opération spécifique exposée par le service. La constitution de traces n'est pas discutée ici. Cette étude fait partie des perspectives du travail décrit dans ce texte.

Value: Community, Criterion, Service ⟶ real

{Value (co, cr ,s) est la valeur de satisfaction du service s pour le critère cr, dans la communauté co. Pré-condition : cr ∈ co.QualityModel. }

Une instance du type Service correspond à la représentation que les communautés ont

d'un service Web.

type Service

⟨ Url : string

{URL à laquelle l'adresse à laquelle service est accessible.}

Adapter : string

{Adapter est l'URL à laquelle l'adaptateur du service est accessible. Cet adaptateur permet d'effectuer la mise en correspondance de l'interface fournie par le service avec celle abstraite de la communauté dans laquelle il est enregistré.}

QoS : {Criterion}

{QoS est l'ensemble des critères associés au modèle de qualité du service s.}

⟩

Dans la figure 6.2, module Selection fournit des fonctions permettant la sélection d'un service concret en fonction des critères de qualité passés en paramètres par le consommateur. Ci-dessous, nous introduisons la fonction Select qui permet le consommateur de sélectionner des services selon un ensemble de critères de qualité, chacun associé à un poids et à une modalité.

Modality : **type** enum {ascendant, descendant}

Weight: **type** real in]0..1]

Restriction: (type ⟶ Boolean)

{Restriction (v) est une fonction booléenne définie sur un type quelconque.}

Select: Community, nb, {⟨Criterion, Modality, Weight, Restriction⟩}⟶ Service

{Soit S = Select (c, nb, {⟨c_1, m_1, p_1, f_1⟩, ...,⟨c_n, m_n, p_n, f_n⟩}. S est le service accessible via la accessible via la communauté c, qui maximise l'ensemble des critères (c_i, i=1..n) selon les modalités associées (m_i, i=1..n) et pondérés par les valeurs p_i. . Chaque service sélectionné a une valeur de satisfaction pour le critère c_i qui vérifie la restriction f_i (i=1..n). L est ordonnée en ordre décroissant des satisfactions des services à l'ensemble des critères ci (i=1..n). La pondération est utilisée pour ramener le choix multi-critère à un choix mono-critère.

Précondition : $\sum_{i=1...n} P_i = 1 \wedge \forall$ i=1..n, $c_i \in$ c.QualityModel}

L'ensemble des critères fourni à la sélection est un sous-ensemble des critères du modèle de qualité de la communauté. Les services qui ne répondent pas à tous les critères de la sélection ne sont pas considérés. La notion de modalité permet de fixer la comparaison d'un ensemble de services selon les valeurs associées à chacun pour un critère de qualité donné.

Par exemple, la modalité ascendant indique que le service s_1 est *meilleur* que s_2 si la valeur de s_1 pour le critère est inférieure à celle de s_2 (le meilleur est le service retourné par la fonction Select). Le poids associé à chaque critère permet de ramener le choix multicritère à un choix monocritère [BAR06]. Finalement, une fonction booléenne est donnée en paramètre et associée à chaque critère. Cette dernière permet d'exprimer une restriction sur les services sélectionnés (par exemple, le temps de réponse doit être inférieur à 10 millisecondes).

Dans la figure 6.2, le module Invocation fournit des fonctions permettant d'exécuter le service sélectionné à travers les opérations offertes dans l'interface du service abstrait de la communauté. Ci-dessous, nous introduisons la fonction Execute qui permet l'exécution d'une opération abstraite d'une communauté par le biais du service retourné par la fonction Select.

Execute: Community, Operation, Message, Service ⟶ Boolean

{Execute(C, O, m, S) est vrai ⇔ La délégation au service S de l'exécution de l'opération abstraite O avec le message d'entrée m a réussi.}

La fonction prend en paramètre les identifiants de la communauté, de l'opération abstraite que le consommateur souhaite exécuter, et le service sélectionné à l'étape précédente. Ce service est en charge d'exécuter la requête du consommateur provenant en termes de l'interface abstraite de la communauté. Le message arrivé passe à travers l'adaptateur associé au service S afin de transformer le message en une représentation qui conforme à celle décrite dans l'interface du service S.

Dans la figure 6.2, Le module Maintenance fournit des fonctions, à destination des fournisseurs de service, leur permettant d'enregistrer/retirer un service à/de la communauté. Ci-après les descriptions de ces fonctions :

Register: Community, Service ⟶ Boolean

{Register(C, S) est vrai ⇔ l'inscription auprès de la communauté C du service S a réussi : le modèle de qualité du service est compatible avec celui de la communauté, et un adaptateur a été généré. Cette inscription rend le service accessible via la communauté.}

Retire: Community, Service ⟶ Boolean

{Retire(C, S) est vrai ⇔ Le service S n'est plus inscrit à la communauté C. Pré-condition : S ∈ C. Service.}

6.4.2 Application au scénario

Nous présentons dans cette section l'application concrète du modèle proposé dans la

section 6.4.1 au scénario décrit plus haut (voir section 6.2.1). Nous nous limitons à la représentation de la communauté Secours dédiée à l'assistance et au transport des personnes (nous faisons abstraction de la représentation qui peut être réalisée, entre autres, en XML). Nous illustrons par quelques exemples l'utilisation de la sélection.

Modality : type enum {ascendant, descendant}
CR1, CR2, CR3 : Criterion
CR1 = ⟨ 'Prix', float, λx •x ∈ [1.00, 80.00], ⟨ 'devise', '$AUD' ⟩ ⟩,
{Les valeurs du critère Prix sont des flottants et comprises entre 1 et 80. Ce sont des devises exprimées en dollar australien.}
CR2 = ⟨ 'TempsDeRéponse', float, λx •x ∈ [0.01, 2.00], ⟨ 'durée', 'millisecondes' ⟩ ⟩,
{Les valeurs du critère sont des durées, exprimées en millisecondes.}
CR3 = ⟨ 'Réputation', 'integer', λx •x {1, 2, 3, 4}, Null ⟩
{Les valeurs du critère Réputation sont des entiers parmi {1, 2, 3, 4}. Il n'y a pas d'information quant à leur nature.}
C : Community = ⟨
C.Name = 'Secours'
C.QualityModel = {CR1, CR2, CR3} {Le modèle de qualité de la communauté.}
C.AbstractInterface = ...
C.Services = {S1, S2} {Les services S1 et S2 sont décrits ci-après.}
⟩
S1 : Service = ⟨
S1.Url = 'www.Ambulance38.com'
S1.Adapter = 'www.Ambulance38.com/FromSecours'
S1.QoS = {CR2, CR3}
{Le modèle de qualité de S1 correspond partiellement à celui de la communauté Secours à laquelle il participe.}
⟩
S2 : Service = ⟨
S2.Url = 'www.RapidesTaxis.com'
S2.Adapter = 'www.RapidesTaxis.com/AdaptateurSecours'
S2.QoS = {CR1, CR2, CR3}
⟩

Nous supposons que l'application Urgence a été conçue de manière à pouvoir interagir avec les communautés, c'est-à-dire que l'interface qu'elle requiert correspond à l'interface abstraite de la communauté d'une part, et d'autre part qu'elle peut, par le biais de l'API OSC,

utiliser la fonction de sélection implantée par la communauté. Des expressions de sélection sont données ci-après. Pour chacun le résultat est calculé sur la base des évaluations de la fonction Value fournies dans le tableau 1. Cette fonction n'est pas applicable avec le critère Prix pour le service S1. Nous faisons ici abstraction du mode d'évaluation de la fonction Value, pour chaque critère et chaque service.

	Prix	TempsDeRéponse	Réputation
S1		0.02	1
S2	10.5	0.04	4

Tableau 6.1 Evaluation de la fonction Value() sur les services S1 et S2, relativement au modèle de qualité de la communauté c.

Select (c, 1, {⟨ CR2, ascendant, 1, λx •x < 1.0 ⟩}) ⟶ [S1]

{Exprime la sélection d'une liste de 1 service, soit s ; s est le service dont la valeur du critère 'TempsDeRéponse' est minimale, parmi ceux accessibles via c et dont la valeur de ce critère est < 1.0. Conformément au tableau 1, S1 est le seul service sélectionné.}

Select (c, 2, {⟨ CR3, descendant, 1, λx • vrai ⟩})) ⟶ [S2, S1]

{Exprime la sélection d'une liste des 2 meilleurs services parmi ceux accessibles via c, selon le critère 'réputation' et la modalité 'descendant'. Cette liste est ordonnée en ordre décroissant de la valeur du critère (S2 est meilleur que S1). }

Select (c, 1, { ⟨ CR3, descendant, 0.8, λx • vrai ⟩
⟨ CR2, ascendant, 0.2, λx • x < 1.0 ⟩}) ⟶ [S1]

{Exprime la sélection multi-critère du meilleur service parmi ceux accessibles via c, selon les 2 critères 'Réputation' avec la modalité descendant, pondéré par 0.8, et 'TempsDeRéponse' avec la modalité ascendant et pondérée par 0.2. Selon le critère 'Réputation' le classement est [S2, S1], par suite S2 (resp. S1) obtient pour ce critère un score de 1 (resp. de 2), tandis que pour le critère 'TempsdeRéponse' le classement est [S1, S2] et le score obtenu par S1 (resp. S2) est 1 (resp. 2). En tenant compte des pondérations données, on obtient pour S1 un score égal à 1.2 alors que le score de S2 est 1.8.}

Select (c, 1, { ⟨ CR1, descendant, 1, λx • x < 5 ⟩}) ⟶ [ ]

{Aucun service n'est sélectionné : le modèle de qualité de S1 n'inclut pas le critère 'Prix', quant à S2 sa valeur pour ce critère ne vérifie par la condition λx • x < 5.}

6.5 Travaux connexes

L'adoption à large échelle de la technologie des services web pose le problème de la

découverte et de la sélection dynamique. Nous présentons ci-dessous des approches visant à résoudre ce problème.

Dans [ZAH05] les auteurs s'intéressent à la sélection dynamique d'un service afin d'adapter la composition à laquelle il participe, au profil de l'utilisateur qui en a demandé l'exécution. Ce profil décrit en particulier le dispositif physique sur lequel la composition s'exécute. Contrairement à notre approche, l'adaptation de l'interface fournie par le service est à la charge du client ou de l'utilisateur final qui a demandé la sélection. Dans [MSB08, et SMY+09] les approches proposées s'appuient sur la notion de communauté des services pour fournir des solutions permettant d'assurer la haute disponibilité des services. Dans ces travaux une communauté de services Web est un moyen pour mettre en œuvre le principe de réplication des données – connu dans le domaine de bases de données – dans l'architecture à base de services.

D'autres approches, décrites ci-dessous, se distinguent par le fait qu'elles s'appuient sur la notion de communauté ou sur des ontologies.

Les communautés : la notion de communauté n'est pas propre au contexte des services web. Dans le domaine des grilles, par exemple, des solutions pour le partage de ressources sur une grille s'appuient sur des communautés [CZA05, FOS06]. A chaque ressource est associé un modèle de qualité contrôlé par le fournisseur de la ressource. Ce principe s'oppose à notre approche où le modèle de qualité est défini par la communauté (les services peuvent le satisfaire, tout ou en partie). Les services sont liés à la communauté par des contrats qui spécifient le niveau de qualité qu'ils s'engagent à offrir. Cette notion est proche de celle modélisée par la fonction Value lorsque son évaluation fournie une valeur donnée par le service et stockée par la communauté. La solution que nous avons choisie pour l'évaluation de la satisfaction d'un service pour un critère est moins restrictive car la fonction d'évaluation des valeurs d'un critère peut être implantée selon plusieurs modes.

WS-catalogNet [PAI04] est un canevas pour l'organisation et l'intégration d'un grand nombre de catalogues. Ici, les catalogues sont rassemblés en un container, appelé aussi communauté, associé à un domaine spécifique et qui expose une catégorie et une description des produits (via des attributs) référencés dans les catalogues. Les fournisseurs, pour rendre disponibles leurs produits, doivent enregistrer leurs catalogues auprès d'une communauté. WS-catalogNet est un service web qui permet la création des communautés, l'adhésion d'un catalogue auprès d'une communauté et la définition des relations entre ces catalogues. Le but est de fournir un modèle pour interroger (en ligne) la communauté au travers des relations entre les différents catalogues de la communauté. Ce canevas permet d'implanter des portails web destinés à des utilisateurs finaux, mais ne permet pas, comme dans l'approche décrite

dans cet article de mettre en relation une application cliente et les services qui correspondent à ses besoins.

Pour [LUO05] une communauté de services web est un moyen pour partager, utiliser et gérer des ressources en s'appuyant sur les services qui sont fournis par la communauté. L'objectif des fournisseurs est d'offrir un cadre pour la qualité de service, conformément aux demandes des clients. Une communauté de services est constituée d'un ensemble d'utilisateurs et de fournisseurs qui interagissent et collaborent les uns avec les autres dans le but de fournir des services et d'échanger des ressources formant ainsi des réseaux. Contrairement à notre approche, une communauté n'est pas associée à un domaine spécifique de services, mais vise plutôt à fournir un cadre pour l'utilisation des applications choisies en fonction de la qualité offerte.

Notre approche est inspirée, en partie, du concept de communauté de services introduit dans [BDS05]. Dans cette approche, un service web peut rejoindre la communauté, même s'il ne répond pas à l'ensemble des besoins couverts par la communauté. Lorsqu'une application cliente désire utiliser un service, celui-ci est sélectionné puis son exécution est déléguée par la communauté auprès du fournisseur de service. Cependant, dans notre travail, nous retournons l'adresse du service (ou de plusieurs services) au client et c'est à lui de contacter le service directement auprès du fournisseur. En outre, dans [BDS05], les valeurs associées au modèle de qualité de chaque service sont fixées de manière statique. Il n'est pas possible, comme dans notre approche, d'en calculer la valeur au moment de la sélection.

Les ontologies : dans cette classe d'approches, les services sont munis chacun d'une description conceptuelle (appelée ontologie). Dans [PAO02] ces descriptions sont exploitées par un algorithme de mise en correspondance, selon un certain degré d'incertitude, entre la requête qui décrit le service requis et les ontologies des services offerts. Il s'agit de sélectionner des services web qui répondent le mieux à une requête formulée en termes des signatures des opérations, mais qui ne prend pas en compte les propriétés non-fonctionnelles des services, comme nous le proposons ici.

Une autre approche à base d'ontologies est proposée dans [BHC03]. La solution s'appuie sur le langage DAML-S [MBD03]. Les auteurs proposent un algorithme de mise en correspondance "*flexible*" entre une service requête et une ontologie DAML-S pour trouver la meilleure combinaison de services web qui satisfait "au mieux" les sorties de la requête et qui exige "le moins possible" d'entrées qui ne sont pas fournies dans la description de la requête. Comme dans [PAO02], les propriétés non fonctionnelles des services ne sont pas prises en compte.

Dans [MAN03], les auteurs mettent en avant la spécification conceptuelle, sous forme d'une ontologie, du processus métier à implanter afin d'effectuer automatiquement (1) la découverte des services web qui peuvent participer au processus métier, et (2) la réalisation des interactions entre les partenaires du processus. Les services eux aussi sont décrits par des ontologies. Un service participe à un processus métier lorsque sa description correspond à une partie de l'interaction. La sélection de services est ici opérée à la conception et non pas dynamiquement à l'exécution comme c'est le cas dans notre approche.

Enfin, dans [GAR06], les auteurs visent à permettre la sélection d'un service sur des critères fonctionnels aussi bien que non fonctionnels. L'approche s'appuie d'une part, sur une extension de WS-policy par une ontologie OWL pour la modélisation des modèles de qualité des services, et d'autre part sur une extension d'UDDI pour la publication et la découverte de services. Cependant, les auteurs ne discutent pas du moment de la sélection (à la conception vs. à l'exécution), ni de la mise en correspondance de l'interface fournie par le service sélectionné avec celle requise par le client.

6.6 Conclusion

Nous avons présenté une architecture multicouche s'intégrant d'une part un modèle de *Communauté de Services* et d'autre part un progiciel nommé *OSC*. L'architecture à offrir un cadre pour : (i) le remplacement ou la substitution de services web au moment de la conception d'une application (de manière statique) et (ii) la sélection de services web au moment de l'exécution d'une composition (de manière dynamique).

Le modèle de communautés et le progiciel OSC permettent :

- Le regroupement d'un nombre potentiellement grand de services répondant au même ensemble de besoins fonctionnels tout en se distinguant par des propriétés non fonctionnelles.

- L'exposition des opérations offertes par les services de la communauté via une interface unique. Une communauté expose une interface abstraite unique ainsi, toute interaction avec un service quelconque de la communauté s'effectue par le biais des opérations décrites dans cette interface.

- L'expression de sélections de un ou de plusieurs services portant sur des critères de qualité exposés par la communauté.

- Une solution au problème de la volatilité des services web. Le concept de communauté

permet également de résoudre le problème de disponibilité de services web. En effet, une communauté peut être vue comme un niveau d'abstraction intermédiaire que les clients utilisent pour construire des applications. Celles-ci sont exprimées en termes de communautés et non plus en termes de services, ce qui permet de substituer un service défaillant par un autre, augmentant ainsi les chances de conclure chaque exécution avec succès.

L'étude rapportée ici ouvre plusieurs perspectives de recherche. La sélection proposée s'appuie sur un modèle de qualité qui doit être enrichi, par exemple pour couvrir un éventail plus large de critères de qualité ou bien pour permettre de réaliser des conversions entre des valeurs d'un même critère exprimées dans différentes unités (monétaires, temporelles, etc.). Dans l'approche que nous avons proposée, si certains services offrent un modèle de qualité qui n'inclut pas tous les critères utilisés dans une sélection, ils ne sont pas considérés dans cette sélection. Cette contrainte doit être relâchée, surtout lorsqu'aucun service enregistré dans la communauté n'est muni d'un modèle de qualité correspondant à tous les critères utilisés dans la sélection. De façon plus générale, nous avons adopté une vision simplifiée du choix multicritère en nous ramenant, par l'introduction de pondération, à un choix monocritère. Une perspective consiste à intégrer des résultats obtenus dans ce domaine (voir par exemple [BAR06]).

Quatrième partie

Conclusion générale

CHAPITRE 7

Conclusion générale

Sommaire

7.1 Résumé des contributions

Notre contribution dans cette thèse s'étend, tant sur le plan théorique que sur le plan pratique. D'une part, nous proposons un canevas *AdaptTer* pour la génération et le déploiement automatique d'un adaptateur entre deux services consommateur et fournisseur. D'autre part, nous proposons une architecture multicouche s'intégrant des *communautés de services* Web et d'un progiciel *OSC*, pour faciliter la substitution de services Web.

Le canevas *AdaptTer*

Alors que notre proposition pour la génération automatique d'adaptateurs, *AdaptTer*, est motivée par un scénario de substitution de services, la proposition est valable pour tout scénario de sélection dynamique de service, que ça soit dans un cadre de substitution ou bien de composition de services Web. Lorsqu'un consommateur substitue son service fournisseur habituel par un autre service, il est nécessaire que son interface soit rendue compatible avec celle du service sélectionné pour que leurs interactions puissent aboutir. Notre proposition pour réconcilier ce problème se déroule en deux temps. Le premier consiste à détecter les incompatibilités entre les interfaces de services. Le deuxième consiste à générer l'adaptateur permettant de résoudre les incompatibilités détectées lors d'interactions entre les services. Nous résumons, ci-après, les contributions principales du canevas proposé:

Modélisation de l'interface des services et des incompatibilités. Les interfaces de services sont modélisées par des automates déterministes en nombre fini d'états (voir chapitre 3), les patrons des incompatibilités mixtes sont formalisées par le biais des expressions logiques sur les automates (voir chapitre 4, section 4.3). Pour chacune des expressions, nous avons démontré sa complétude vis-à-vis de l'incompatibilité qu'elle modélise. Nous avons ainsi prouvé que les différentes expressions proposées s'excluent mutuellement.

Détection des incompatibilités. La détection des incompatibilités entre deux interfaces est assurée par un algorithme qui compare ces dernières en s'appuyant sur les expressions qui modélisent les incompatibilités. L'algorithme retourne les localisations exactes de toutes les incompatibilités mixtes entre les deux interfaces comparées. Cet algorithme vient en complément des travaux effectués dans le cadre du projet auquel s'inscrit ma thèse [AB08]. Dans ces travaux, l'algorithme proposé retourne des incompatibilités dues à *l'ajout, la suppression,* et *la modification* des opérations.

Génération des adaptateurs à bases des opérateurs de résolution. La génération des adaptateurs s'appuie sur la composition automatique des opérateurs de résolution prédéfinis et exprimés sous forme d'automates configurables (voir chapitre 4, section 4.4). Pour chaque patron d'incompatibilité proposé, nous avons mis en place un opérateur de résolution. Celui-ci est utilisé et configuré automatiquement - en fonction des caractéristiques de l'incompatibilité concernée - par l'algorithme de génération des adaptateurs. L'algorithme retourne l'automate qui spécifie le fonctionnement et le comportement de l'adaptateur destiné à réconcilier les interactions entre les services en question.

Mise en œuvre des adaptateurs avec la technologie CEP. Une originalité de notre approche est la mise en œuvre d'adaptateurs avec la technologie de traitement des événements complexes *CEP* [TFDB08]. L'adaptateur généré sous forme d'un automate, suffisamment détaillé, est transformé en requêtes continues pour la technologie *Coral8* [CCL08] et déployé dans le moteur *CEP* proposé par celle-ci. La transformation s'effectue à l'aide des composants, cartouches, étant conçus pour générer des requêtes continues pour la technologie *Coral8* (voir chapitre 5).

Mise en place d'un outil opérationnel. Le canevas proposé est enrichi avec un support visuel permettant de contrôler graphiquement les différentes étapes de la mise en œuvre des adaptateurs, de la détection des incompatibilités jusqu'à la génération et le déploiement d'adaptateurs. Ce support rend le canevas opérationnel et destiné même aux développeurs des adaptateurs qui n'ont pas nécessairement des connaissances sur la technologie *CEP*, et cela grâce aux IHM proposées. L'outil a été démontré à plusieurs reprises, notamment dans [TFDB08].

Architecture multicouche à bases de communautés et OSC

Lorsqu'un consommateur substitue son service fournisseur habituel par un autre, le canevas *AdaptTer* permet de décharger le consommateur des détails d'adaptation de son interface à celle du nouveau service, et cela par génération automatique de l'adaptateur destiné à réconcilier les interactions entre les deux services. Cependant, la tâche de sélection

du nouveau service, ainsi que la liaison à celui-ci reste toujours à la charge du concepteur du service consommateur. Ce qui rend finalement la substitution de services une tâche manuelle.

L'objectif de l'architecture multicouche que nous proposons dans le chapitre 6 est de permettre une sélection flexible et une liaison transparente au service substitut. Pour ce faire, notre solution met l'accent sur une couche d'abstraction intermédiaire placée entre les consommateurs et les fournisseurs. La notion de communauté de services a été mise en place pour établir le rapport entre la couche abstraite d'une part et les consommateurs et les fournisseurs d'autre part. Nous résumons, ci-après, les contributions principales de l'architecture proposée :

Communautés de services. Le principe d'une *communauté de services* consiste à regrouper des services qualifiés de concrets répondant aux mêmes besoins fonctionnels, et de les présenter par le biais d'une interface commune qualifiée d'abstraite. Pour un consommateur, seule l'interface abstraite de la communauté est visible. Ainsi, les interactions entre le consommateur et la communauté sont réalisées en termes de cette interface. Etant donné qu'une interface abstraite n'implémente pas sa fonctionnalité, les requêtes de consommateur sont déléguées à l'un des services concrets inscrits auprès de la communauté. Lors de l'inscription d'un service auprès de communauté, des tests d'équivalence entre l'interface abstraite de la communauté et celle du service sont réalisés. Dans le cas ou les deux interfaces sont équivalentes, le service peut rejoindre la communauté sans avoir besoin d'adaptation. Sinon, un adaptateur est généré et enregistré dans le registre spécifique des adaptateurs au sein de la communauté. Cet adaptateur a pour rôle de transformer les interactions provenant d'un consommateur d'une manière adaptée à l'interface abstraite de la communauté en des interactions adaptées à l'interface du service concret à lequel sont déléguées les requêtes du consommateur.

Progiciel OSC. Des fonctions de sélection et de substitution de services sont offertes au consommateur par le biais du progiciel *OSC*. Dès lorsqu'un consommateur substitue un service concret par un autre, les interactions entre le consommateur et le service concret sont interceptées par l'adaptateur activé par le *OSC*, et chargé de réconcilier les interactions entre les deux services. Ainsi, la substitution est effectuée de façon transparente pour le consommateur qui continue à interagir avec la communauté en termes de son interface abstraite.

7.2 Limites et perspectives envisagées

La réalisation de notre canevas *AdapTer* nous a permis de dégager plusieurs perspectives

de travail. Nous avons démontré la faisabilité de notre approche pour la détection des incompatibilités et la génération des adaptateurs sur un ensemble des incompatibilités mixtes que nous considérerons primitives dans les interactions entre des services. Cependant des incompatibilités plus complexes peuvent être étudiées, par exemple *l'exécution de plusieurs opérations différentes deux à deux d'un service consommateur versus l'exécution de plusieurs opérations différentes deux à deux également d'un service fournisseur*. L'une des perspectives de cette thèse est de formaliser les expressions qui peuvent modéliser et résoudre ce type des incompatibilités.

Dans le cas où une incompatibilité non reconnue par l'algorithme de détection, un concepteur d'adaptateur peut intervenir en éditant le modèle graphique proposé par le canevas, et en proposant sa propre solution. Puis, lors de la génération de l'adaptateur, les propositions du concepteur sont prises en compte.

Nous envisageons de proposer une approche générique pour l'adaptation des interactions de services combinant les aspects structurels, comportementaux, et sémantiques d'interfaces (en intégrant par exemple les travaux décrits dans [MRI07] et [AB08]).

Dans le canevas *AdaptTer*, l'adaptateur généré entre deux services a été mis en œuvre dans une infrastructure *CEP* en transformant son automate en des requêtes continues pour la technologie *coral8*. L'une des perspectives sur le plan pratique de cette thèse est de transformer l'automate de l'adaptateur en des représentations *BPEL*. L'objectif de cette tâche est d'effectuer de test de monté en charge de l'adaptateur généré dans ses deux versions *CEP* et *BPEL*. Cela nous permettra d'effectuer une étude comparative entre ces deux technologies (*CEP* et *BPEL)* dans le but de recommander celle la plus performante en termes du temps et d'efforts pour la mise en œuvre des adaptateurs de services Web.

En ce qui concerne notre architecture multicouche pour la substitution de services, plusieurs pistes de recherche restent ouvertes. Une de ces pistes concerne la construction automatique des communautés. Un agent logiciel intelligent peut être envisagé pour initier la construction des communautés, et pour inviter les services Web classiques à rejoindre les communautés.

BIBLIOGRAPHIE

[AB08] A. Aït-Bachir. ArchiMed : un canevas pour la détection et la résolution des incompatibilités des conversations entre services Web. Thèse de doctorat en informatique, Université Joseph Fourier - Grenoble1, 2008.

[ABF07] A. Aït-Bachir and M.C. Fauvet. Un médiateur pour la réconciliation de conversations entre services web. In *INFORSID*, pages 537-552, Perros-Guirec, France, 2007.

[ACKM04] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. Web Services - Concepts, Architectures and Applications. Springer-Verlag, pages: 8-30, Berlin 2004.

[AF03] V. R. Aragão and A. A. Fernandes. Conflict resolution in web service federations. In *Proceedings of the International Conference on Web Services-Europe, ICWS-Europe*, LNCS, pages 109-122, 2003.

[AMP03] V. De Antonellis, M. Melchiori, and P. Plebani. An approach to web service compatibility in cooperative processes. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops, SAINT Workshops, page 95*, IEEE Computer Society, USA, 2003.

[ATP06] G. Athanasopoulos, A. Tsalgatidou, and M. Pantazoglou. Interoperability among heterogeneous services. In *Proceedings of the IEEE International Conference on Services Computing, SCC*, pages 174-181, IEEE, USA, 2006.

[BAR06] D. B. CLARO. SPOC - Un canevas pour la composition automatique de services web dédiés à la réalisation de devis. *Thèse de Doctorat*, Ecole Doctorale d'Angers, 2006.

[BBG+04] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. *Architecting Systems with Trustworthy Components, volume 3938 of LNCS*, chapter Towards an Engineering Approach to Component Adaptation, pages 193--215. Springer Verlag, 2004.

[BCG+05] B. Benatallah, F. Casati, D. Grigori, H.R. Motahari-Nezhad, and F. Toumani. Developing adapters for web services integration. In *Proceedings of the 17th International Conference on Advanced Information System Engineering, CAiSE*, pages 415- 429, Porto, Portugal 2005. Springer-Verlag

[BCH05] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proceedings of the 14th international conference on World Wide Web, WWW*, pages 148- 159, Chiba, Japan, 2005.

[BCPV04] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. In *Proceedings of the First International Workshop on Web Services and Formal Methods Formalizing web service choreographies, WSFM, pages* 73- 94, 2004.

[BCT06a] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data Knowledge Engineering*, 58(3):327- 357, 2006.

[BCT+06b] B.Benatallah, Fabio Casati, Farouk Toumani, Julien Ponge, and Hamid R. Motahari Nezhad. Service mosaic: A model-driven framework for web services life-cycle management. *IEEE Internet Computing*, 10(4):55- 63, 2006.

[BDDM05] B. Benatallah, R. Dijkman, M. Dumas, and Z. Maamar. Service Composition: Concepts, Techniques, Tools and Trends. In *Service-Oriented Software System Engineering: Challenges and Practices.* IDEA Group, Hershey, PA, pp. 48-66

[BDS05] B. Benatallah, M. DUMAS, Q. SHENG. Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services. *Distributed and Parallel Database*, vol. 17, no 1, 2005, p. 5-37, Springer Science, Business Media.

[BHC03] B. Benatallah, M.S. Hacid, C. Rey, F. Toumani. The SemanticWeb-ISWC 2003, chapitre Request Rewriting-Based Web Service Discovery, p. 242-257, LNCS, Springer Berlin-Heidelberg, 2003.

[BL04] S. Bowers and B. Ludäscher. An ontology-driven framework for data transformation in scientific workows. *International Workshop on Data Integration in the Life Sciences*, DILS, pages 1- 16. Springer, 2004.

[BLQ+03] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, USA, 2003.

[BMT+07] D. Benslimane, Z. Maamar, Y. Taher, M. Lahkim, M.C. Fauvet, and M. Mrissa. A multi-layer and multiperspective approach to compose web services. In *Proceedings of the 21st International Conference on Advanced Networking and Applications, AINA*, pages: 31-37, Niagara Falls, Canada. 2007.

[BP06] A. Brogi and Razvan Popescu. Automated generation of bpel adapters. In *Proceedings of the 4th International Conference on Service Oriented Computing, ICSOC*, pages 27- 39, Chicago, IL, USA 2006.

[BSBM04] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are two web services compatible? In *Proceedings of the 5th International Conference on Technologies for E-Services*, pages 15- 28, Toronto, Canada, 2004. Springer Verlag.

[BSD03] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40- 48, 2003.

[Bus03] C. Bussler. Semantic web services: Reflections on web service mediation and composition. In *Proceedings of the 4th International Conference on Web Information Systems Engineering, WISE*, pages: 253 - 260.

[CCC07] K. M. Chandy, M. Charpentier, and A. Capponi. Towards a theory of events. In *Proceedings of the 2007 inaugural international conference on Distributed event-based system, DEBS*, pages 180-187, Toronto, Ontario, Canada 2007.

[CCL08] Systems, Coral8 CCL Reference Version 5.4, http://www.coral8.com/-system/files/assets/pdf/current/Coral8CclReference.pdf, 2009.

[CD05] L. Cabral and J. Domingue. Mediation of Semantic Web Services in IRS-III. In *Proceedings of the Workshop on Mediation in Semantic Web Services in conjunction with the 3rd International Conference on Service Oriented Computing*, Amsterdam, The Netherlands, (2005).

[CAH05] D. B. Claro, P. Albers, and J. K. Hao. Selecting web services for optimal composition. In *Procedings of the 2005 International Conference on Web Services, ICWS*, 2005.

[CHA06] M. Chanliau. Web services security: What's required to secure a service oriented architecture, http://www.oracle.com/technology/tech/standards/pdf/-security.pdf, 2006.

[CLB08] H. S. Chae, J. Lee, and J. H. Bae. An approach to checking behavioral compatibility between web services. International Journal of Software Engineering and Knowledge Engineering, 18(2):223- 241, 2008.

[CZA05] K. Czajkowski, I. Foster, C. Kesselman. Agreement-based resource management», Proceedings of the IEEE, vol. 93, March 2005, p. 631- 643.

[DGNT04] S. Derong, Y. Ge, Y. Nan, and N. Tiezheng. Heterogeneity resolution based on ontology in web services composition. In *Proceedings of the Workshop on E-Commerce Technology for Dynamic E-Business*, pages 274- 277, 2004.

[DHM+04] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. *In Proceedings of the Thirtieth international conference on Very large data bases, VLDB*, pages: 372- 383, Toronto, Canada, 2004.

[DSW06] M. Dumas, M. Spork, and K. Wang. Adapt or perish : Algebra and visual notation for service interface adaptation. In *Procedings of the 4th International Conference on Business Process Management, BPM*, number 4102 in LNCS, pages 65-80, Vienna, Austria, September 2006. Springer Verlag.

[ELP06] M. D. Ernst, R. Lencevicius, and J. H. Perkins. Detection of web service substitutability and composability. *International Workshop on Web Services - Modeling and Testing, WS-MaTe*, pages 123-135, Palermo, Italy, 2006.

[FDD05] M.C. Fauvet, H. Duarte, M. Dumas, B. Benatallah. Handling Transactional Properties in Web Service Composition, LNCS S.-V., Ed., WISE'05, vol. 3806, *The 6th International Conference on Web Information Systems Engineering*. New York City, New York, 20-22 Nov 2005, p. 273-289.

[FDTB07] M.C. Fauvet, H. Duarte, Y. Taher, and D. Benslimane. Sélection dynamique de services web - une approche à base de communautés. In *INFORSID*, Perros-Guirec, France, pages 505-520, 2007.

[FUC04] M. Fuchs. Adapting web services in a heterogeneous environment. In *Procedings of the 2004 International Conference on Web Services, ICWS*, pages: 656- 664, California, USA, 2004.

[FUMK04] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *Procedings of the 2004 International Conference on Web Services, ICWS*, pages 738- 741. California, USA, 2004.

[FHH+01] D. Fensel, F. V. Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider. Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38- 45, 2001.

[FOS06] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayer, X. Zhang. Virtual Clusters for Grid Communities. *CCGRID, Sixth IEEE International Symposium on Cluster Computing and the Grid*, Singapore, IEEE Computer Society, 16-19 May 2006, p. 513-520.

[GAR06] D.-Z.-G GARCIA, M.-B.-F. DE TOLEDO. Semantics-enriched QoS policies for web service interactions. In *Proceedings of the 12th Brazilian symposium on Multimedia and the web*, New York, NY, USA, 2006, ACM Press, pages: 35–44.

[GKG03] D. Gouscos, M. Kalikakis, and P. Georgiadis. An approach to modeling web service qos and provision price. In *Procedings of the 4th International Conference on **Web Information Systems Engineering Workshops**, WISEW*, pages: 121-130, 2003.

[GÖD29] K. Gödel. Complétude du calcul logique. Thèse de doctorat en Mathématiques, 1929.

[HCM+05] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. Wsmx - a semantic service-oriented architecture. In I. C. Society, editor, *ICWS*, pages 321–328. IEEE Computer Society, 2005.

[HMU00] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Languages, and Computation (2nd Edition). Addison Wesley, November 2000.

[HMO05] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. Wsmx - a semantic service-oriented architecture. In *proceeding of the international Conference on Web Service, ICWS*, pages 321–328. IEEE Computer Society, 2005.

[HOF05] A. H. M. Ter Hofstede. Yawl : yet another workow language. *Information Systems*, 30:245- 275, 2005.

[HOW95] HOWES T., KILLE S., YEONG W., « Lightweight Directory Access Protocol », www.ietf.org/rfc/rfc17777.txt, 1995.

[JHKK04] J. y. Jung, W. Hur, S. Kang, and H. Kim. Business process choreography for b2b collaboration. *IEEE Internet Computing*, 8(1):37- 45, 2004.

[JMS+08] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Cetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming sql

standard. *In Proceedings of the international conference on Very large data bases, VLDB*, 1(2) :1379-1390, 2008.

[KBH+04] T. Kawamura, J. Blasio, T. Hasegawa, M. Paolucci, and K. Sycara. Public Deployment of Semantic Service Matchmaker with UDDI Business Registry. In Proceedings of *the 3rd International Semantic Web Conference (ISWC2004)*, LNCS 3298, pp. 752-766, 2004. Springer- Verlag, Berlin.

[KKL04] S. Kalepu, S. Krishnaswamy, and S. W. Loke. Reputation = f(user ranking, compliance, verity). In *Procedings of the 2004 International Conference on Web Services, ICWS*, pages: 200-2007, California, USA, 2004.

[KLZ02] S. Krishnaswamy, S. W. Loke, and A. B. Zaslavsky. Application run time estimation: a quality of service metric for web-based data mining services. In *Procedings* of the ACM Symposium on Applied Computing, SAC, pages 1153-1159, 2002.

[KML06] P. Kaminski, H. Müller, and M. Litoiu. A design for adaptive web service evolution. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, SEAMS*, pages 86- 92, New York, NY, USA, 2006. ACM.

[KS96] V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: A context-based approach. *VLDB Journal*, 5:276- 304, 1996.

[KSPBC06] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An aspect-oriented framework for service adaptation. In *Proceedings of the 4th International Conference on Service Oriented Computing, ICSOC,* pages 15- 26, Chicago, IL, USA 2006.

[LGL06] B. Lin, N. Gu, and Q. Li. A requester-based mediation framework for dynamic invocation of web services. In *Proceedings of the IEEE International Conference on Services Computing, SCC*, pages 445- 454, IEEE, USA, 2006.

[LJ03] Y. Li and H. V. Jagadish. Compatibility determination in web services. *In ICEC Workshop on on E-Government and Web Services*, 2003.

[LNM+01] T. Ledoux, J. Noyé, J-M Menaud, M. Blay-Fornarino, D. Caromel, E. Bruneton, T. Coupaye, D. Hagimont, and M. Riveill. Etat de l'art sur l'adaptabilité. Technical Report Livrable D1.1, RNTL ARCAD, December 2001.

[LNZ04] Y. Liu, A. H. Ngu, and L. Z. Zeng. Qos computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, WWW Alt.*, pages 66-73, New York, NY, USA, 2004. ACM Press.

[LUO05] Z. LUO, J.-S. LI. A Web Services Provisioning Optimization Model in a Web Services Community. *ICEBE, IEEE International Conference on e-Business engineering*, Beijing, China, IEEE Computer Society, 18-20 October 2005, p. 689-696.

[LSZ+06] F. Liu, Y. Shi, L. Zhang, L. Lin, and B. Shi. Analysis of web services composition and substitution via ccs. Data Engineering Issues in E-Commerce and Services, DEECS, pages 236- 245, 2006.

[MAN03] D.J. Mandell, S.-A. Mcilraith. The Semantic Web - ISWC 2003, chapitre Adapting BPEL4WS for the Semantic Web : The Bottom-Up Approach to Web Service Interoperation, p. 224-241, LNCS, Springer Berlin-Heidelberg, 2003.

[MB05] B. Medjahed and A. Bouguettaya. A dynamic foundational architecture for semantic web services. *Distributed and Parallel Databases*, 17(2):179- 206, 2005.

[MBD03] D. Martin, M. Burstein, G. Denker, J. Hobbs, L. Kagal, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. DAML-S (and OWL-S) 0.9 draft release, 2003.

[MCS+05] A. Mocan, E. Cimpian, M. Stollberg, F. Scharffe, and J. Scicluna. WSMO mediators. WSMO Working Draft D29v0.2, DERI, December 2005.

[MIL80] R. Milner. A calculus of communicating systems, volume 92. 1980.

[MMGF06] A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing compatibility of bpel processes. In *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services, AICT-ICIW*, page 147, Washington, DC, USA, 2006.

[MPC01] M. Mecella, B. Pernici, and P. Craca. Compatibility of e - Services in a cooperative multi-platform environment. In *Proceedings of the Second International Workshop on Technologies for E-Services, TES*, pages 44- 57, London, UK, 2001. Springer-Verlag.

[MRI07] M.Mrissa. Médiation Sémantique Orientée contexte pour la composition de services Web. Thèse de doctorat en informatique, Université Claude Bernard Lyon 1, 2007

[MSB08] Z. Maamar, Q. Z. Sheng, D. Benslimane. Sustaining Web Services High-Availability Using Communities. In *Proceedings of the The Third International Conference on Availability, Reliability and Security, ARES*, pages: 834-841, Barcelona, Spain, 2008.

[MVR+04] J. Miller, K. Verma, P. Rajasekaran, A. Sheth, R. Aggarwal, and K. Sivashanmugam. WSDL-S: Adding Semantics to WSDL - White Paper. *Technical report, Large Scale Distributed Information Systems*, 2004. http://lsdis.cs.uga.edu/library/ download/wsdl-s.pdf.

[NBCT06] H. R. Motahari-Nezhad, B. Benatallah, F. Casati, and F. Toumani. Web services interoperability specifications. *IEEE Computer*, 39(5) 24- 32, 2006.

[NBM+07] H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th international conference on World Wide Web, WWW*, pages 993- 1002, New York, NY, USA, 2007.

[NG09] http://search.cpan.org/dist/Algorithm-NGram/. Last visit May 2009.

[NM03] S. Narayanan and S. McIlraith. Analysis and simulation of web services. *Comput. Networks*, 42(5) :675–693, 2003.

[NVS+06] M. Nagarajan, K. Verma, A. P. Sheth, J. Miller, and J. Lathem. Semantic interoperability of web services - challenges and experiences. In *Procedings of the 2004 International Conference on Web Services, ICWS*, pages: 373- 382, 2006.

[OAS07] OASIS UDDI, « Universal Description, Discovery and Integration version 3.0», http ://www.uddi.org, 2009.

[PAI04] H.-Y. Paik, B. Benatallah, F. Toumani. WS-CatalogNet : Building Peer-to-Peer e-Catalogs, *FQAS 2004, 6th International Conference on Flexible Query Answering Systems*. Lyon, France, June 24-26 2004, p. 256-269.

[PBM03] P. F. Pires, M. R. F. Benevides, and M. Mattoso. Mediating heterogeneous web services. In *Proceedings of the 2003 Symposium on Applications and the Internet, SAINT, pages:* 344-347, IEEE Computer Society, USA, 2003.

[PF04] S. R. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Proceedings of the 5th ACM International Conference on Middleware*, pages 331- 351, 2004.

[PAO02] M. Paolucci, T. Kawamura, T.R. Payne, K.P. Sycara. Semantic Matching of Web Services Capabilities, ISWC 2002, vol. 2342 de Lecture Notes in Computer Science, *First International Semantic Web Conference*, Sardinia, Italy, Springer, June 9-12 2002, p. 333-347.

[RC04] U. Radetzki and A. B. Cremers. IRIS: A framework for mediator based composition of service-oriented software. In *Procedings of the 2004 International Conference on Web Services, ICWS*, pages: 752 - 756, California, USA, 2004.

[Ref99] Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[SL04] B. Spencer and S. Liu. Inferring data transformation rules to integrate semantic web services. In *Proceedings of the International Semantic Web Conference, IWSC*, pages 456- 470, 2004.

[SMY+09] Q. Z. Sheng, Z. Maamar, J. Yu, A. H.H. Ngu: Robust Web Services Provisioning through On-Demand Replication. In *Proceedings of the Third International United Information Systems Conference, UNISCON*,pages: 4-16, Sydney, Australia, 2009.

[TABFB09] Y. Taher, A. Aït-Bachir, M.C. Fauvet, and D. Benslimane. Diagnosing Incompatibilities in Web service Interactions for Automatic Generation of Adapters. In *Proceedings of the 23st International Conference on Advanced Networking and Applications, AINA,* Bradford, UK, 2009.

[TBFM06] Y. Taher, D. Benslimane, M.C. Fauvet, and Z. Maamar. Towards an approach forweb services substitution. In *Database Engineering and Applications Symposium, IDEAS*, pages: 166- 173, 2006.

[TD05] A. Tolk and S. Y. Diallo. Model-based Data Engineering for Web Services. *IEEE Internet Computing*, 9(4), July/August 2005.

[TFDB08] Y. Taher, M.C. Fauvet, M. Dumas, and D. Benslimane. Using CEP technology to adapt messages exchanged by web services. In *Proceedings of the 17th international conference on World Wide Web, WWW*, pages 1231- 1232, Beijing, China, 2008.

[WBC06] S. K. Williams, Steven A. Battle, and J. E. Cuadrado. Protocol mediation for adaptation in semantic web services. *The Semantic Web: Research and Applications*, Volume 4011/2006 pages 635- 649, 2006.

[WDR06] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 international conference on Management of data, SIGMOD*, pages 407-418, New York, NY, USA, 2006. ACM.

[WFMN04] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. In *Proceedings of the 2004 IEEE International Conference on e- Technology, e-Commerce and e-Service, (EEE'04)*, pages 359- 368,Washington, DC, USA, 2004.

[WS03] Y. Wang and E. Stroulia. Flexible interface matching for web-service discovery. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, WISE*, pages: 147-156, Washington, DC, USA, 2003.

[WW05] J. Wu and Z. Wu. Similarity-based web service matchmaking. In *Proceedings of the 2005 IEEE International Conference on Services Computing, SCC*, pages 287-294, Washington, DC, USA, 2005.

[YZL06] T. Yu, Y. Zhang, and K.J. Lin. Modeling and measuring privacy risks in qos web services. In *Proceedings of the Third International Conference Enterprise Computing, E-Commerce, and E-Services, CEC-EEE*, page: 4-4, San Francisco, CA, USA, 2006

[ZAH05] W. Zahereddine, Q.H Mahmoud. A Framework for Automatic and Dynamic Composition of Personalized Web Services, AINA, 19th International Conference on Advanced Information Networking and Applications, Taipei, Taiwan, IEEE Computer Society, 28-30 March 2005, p. 513-518.

[ZYGW06] J. Z. Zhang, S. j. Yu, X. k. Ge, and G. Wu. Automatic web service composition based on service interface description. In *Proceedings of the International Conference on Internet Computing*, pages 120 - 126, 2006.