
Ecole doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)
(Grenoble INP / Université Joseph Fourier - Grenoble 1)

ArchiMed : un canevas pour la détection et la résolution des incompatibilités des conversations entre services web

THÈSE

présentée et soutenue publiquement le 28 novembre 2008

pour l'obtention du

Doctorat de l'Université Joseph Fourier – Grenoble 1

(spécialité informatique)

par

Ali Aït-Bachir

Directrice de thèse : Marie-Christine Fauvet, Professeur UJF

Composition du jury

<i>Président :</i>	Djamal Benslimane	Université Claude Bernard, LIRIS
<i>Rapporteurs :</i>	Anne Doucet	Université Pierre et Marie Curie, LIP6
	Claude Godart	Université Henri Poincaré Nancy 1, ESSTIN
<i>Examineurs :</i>	Marie-Christine Fauvet	Université Joseph Fourier, LIG
	Djamal Benslimane	Université Claude Bernard, LIRIS
	Marlon Dumas	Université de Tartu, Institute of Computer Science

Remerciements

Je tiens avant tout à remercier Marie-Christine Fauvet pour m'avoir proposé de travailler sur ce sujet de recherche passionnant. Je lui suis reconnaissant d'avoir su partager son expérience et ses qualités de chercheur. Merci d'avoir été disponible et de me conseiller.

Je remercie Marlon Dumas pour l'accueil chaleureux auquel j'ai eu droit durant mon séjour à Tartu en Estonie. Merci d'avoir partagé ta vision de la recherche.

Je remercie vivement les membres du jury pour avoir accepté d'évaluer mes travaux. En particulier, merci à Anne Doucet, Professeur à l'université Pierre et Marie Curie, ainsi qu'à Claude Godart, Professeur à l'Université Henri Poincaré Nancy 1, d'avoir pris le temps de rapporter ma thèse et pour l'intérêt qu'ils ont manifesté pour ce travail. Je remercie également Djamel Benslimane, Professeur à l'université Claude Bernard Lyon1 d'avoir examiné ma thèse.

Je remercie mes collègues de l'équipe MRIM, ainsi que d'autres collègues des autres équipes du laboratoire LIG avec lesquels j'ai partagé de bons moments d'échanges aussi bien sur la plan humain que scientifique.

Mes remerciements vont aussi à tous ceux qui m'ont permis d'effectuer ce travail dans de bonnes conditions : merci à tous mes amis d'avoir été là, votre bonne humeur m'ont permis de surmonter mes difficultés.

Merci à mes parents pour m'avoir encouragé et aider à mener ma vie d'étudiant si longtemps et si loin de vous qui êtes en Algérie. Votre soutien constant à travers ces longues années m'a été précieux. Merci à mes frères et sœur : Nassim, Lotfi et Radia :)

Je remercie enfin ma femme : merci Samira d'être là. Merci pour m'avoir accompagné, encouragé, écouté et compris durant des moments pas toujours faciles. J'espère que nous pourrons concrétiser nos rêves.

Table des matières

Partie I	Contexte de la recherche et état de l'art.	3
-----------------	---	----------

Chapitre 1

Contexte et problématique

1.1	Introduction	5
1.2	Concepts fondamentaux	8
1.2.1	Communication par envois et réceptions de messages	8
1.2.2	Interfaces des services web	9
1.2.3	Composition de services web : orchestration et chorégraphie	12
1.2.4	Conformité et équivalence des interfaces	14
1.3	Problématique : incompatibilités des interfaces	17
1.3.1	Comportement et structure	18
1.3.2	Ajout d'une opération	19
1.3.3	Suppression d'une opération	20
1.3.4	Modification des opérations dans une séquence	21
1.4	Objectifs de la thèse et approche ArchiMed	22
1.5	Plan de la thèse	23

Chapitre 2

État de l'art

2.1	Introduction	25
2.2	Tests de conformité et résolution des incompatibilités	26
2.2.1	Comportements et structures	27
2.2.2	Compatibilité à la conception d'une composition de services . . .	30
2.2.3	Résolution à l'exécution	39
2.3	Changements des interfaces	46
2.3.1	Patrons de changement	46
2.3.2	Évolution des instances de processus en cours d'exécution . . .	49
2.3.3	Versions d'une interface fournie	52
2.4	Synthèse	58
2.4.1	Structure et comportement	58
2.4.2	Détection	59
2.4.3	Résolution	60
2.4.4	Tableaux récapitulatifs	62
2.5	Conclusion	66



Partie II Proposition d'un canevas pour la réconciliation de conversations par génération automatique de médiateurs. 67

Chapitre 3

Modélisation et principes de la solution

3.1	Introduction	69
3.2	Modélisation des interfaces des services	70
3.2.1	Modélisation du comportement	70
3.2.2	Modélisation de l'interface structurelle	72
3.3	Principes généraux de la proposition	73
3.3.1	Principe de détection des incompatibilités par détection des différences	73
3.3.2	Simulation des interfaces et maintien de la conformité	76
3.3.3	Principe de la résolution par médiation	78
3.4	Conclusion	79

Chapitre 4

Détection

4.1	Introduction	81
4.2	Incompatibilités élémentaires	82
4.2.1	Notations pour l'expression des incompatibilités	82
4.2.2	Suppression d'une opération	83
4.2.3	Ajout d'une opération	86
4.2.4	Modification d'une opération	88
4.3	Exclusion mutuelle et complétude des expressions	89
4.3.1	Exclusion mutuelle des expressions de détection	90
4.3.2	Etude de la complétude	92
4.4	Algorithme de détection des incompatibilités	93
4.4.1	Principe de l'algorithme de détection	93
4.4.2	Détail de l'algorithme de détection	95

4.4.3	Etude de la complexité de l'algorithme	97
4.5	Conclusion	99

Chapitre 5 Résolution
--

5.1	Introduction	101
5.2	Incompatibilités automatiquement résolubles	102
5.2.1	Suppression d'un choix d'opération dans une alternative	102
5.2.2	Suppression de l'acquiescement d'une opération	103
5.3	Résolution des incompatibilités	104
5.3.1	Cas de suppression d'un choix d'opération	105
5.3.2	Cas de suppression d'un acquiescement d'opération	106
5.3.3	Algorithme de résolution des incompatibilités	107
5.4	Génération automatique de médiateurs	109
5.4.1	Interactions via des médiateurs	109
5.4.2	Utilisation de médiateurs	110
5.4.3	Sélection des médiateurs	113
5.5	Conclusion	115

Chapitre 6 Mise en œuvre d'ArchiMed
--

6.1	Introduction	117
6.2	Mise en œuvre du prototype	118
6.2.1	Architecture logicielle	118
6.2.2	Détails d'implantation	119
6.2.3	Diagramme de classes de la détection des incompatibilités	122
6.3	Scénarios d'utilisation de ArchiMed	123
6.3.1	Point d'entrée de ArchiMed	123
6.3.2	Détection des incompatibilités entre deux interfaces	124
6.3.3	Mesure de similarité entre les interfaces	126
6.3.4	Détection des incompatibilités automatiquement résolubles	127
6.4	Tests sur la détection des incompatibilités	128
6.4.1	Construction de la base de tests	129

6.4.2	Tests de similarité des interfaces	129
6.4.3	Etudes comparatives	130
6.5	Conclusion	134

Chapitre 7	
Conclusion générale et perspectives	

7.1	Contributions de la thèse	135
7.2	Limites et perspectives	136

Annexes	139
Annexe A Démonstrations de l'exclusion mutuelle	139
Annexe B Exemple description en WSDL d'un service web	141
Annexe C Intervalle de définition de la mesure de similarité moyenne	145
Annexe D Transformation de modèles SCXML vs BPEL	147

Bibliographie	149
----------------------	------------

Table des figures

1.1	Communication entre des services web	9
1.2	Interface fournie du service Entrepôt (notation UML : diagramme d'activités)	10
1.3	Interface fournie versus requise	12
1.4	Chorégraphie et orchestration de services web	13
1.5	Conformité des interfaces fournie et requise	15
1.6	Equivalence contextuelle des interfaces fournies	16
1.7	Changement de la structure des messages	18
1.8	Ajout d'une opération	19
1.9	Suppression d'une opération	20
1.10	Modification d'une opération	21
2.1	Adaptateur : plusieurs envois de messages <i>versus</i> réception unique de message	28
2.2	Représentation en automate d'un processus de commande	34
2.3	Similarité : structure de graphe et sémantique de comportement	37
2.4	Mesure de similarité entre deux automates	39
2.5	Architecture de l'annuaire de services web	40
2.6	Patron d'insertion d'un fragment de processus	47
2.7	Modèle d'un processus et instances du modèle du processus	50
2.8	Chaîne d'adaptateurs après n versions	53
2.9	Arbre de versions du modèle de processus wt [71]	55
3.1	Modélisation en LTS de l'interface d'un service	71
3.2	Exemple d'incompatibilités	74
3.3	Détection des différences entre deux interfaces fournies	75
3.4	Simulation des interfaces et conformité	77
3.5	Principe de la médiation de la conversation	78
3.6	Diagramme de séquences d'une conversation par médiation	78
4.1	Suppression d'une opération alternative	84
4.2	Suppression d'une opération dans une séquence	85

4.3	Ajout d'une opération dans une alternative	86
4.4	Ajout d'une opération dans une séquence	87
4.5	Cas de détection de la modification d'opération	88
4.6	Principe de progression de l'algorithme	94
4.7	Algorithme de détection des incompatibilités élémentaires	96
5.1	Suppression d'un terme d'une alternative	103
5.2	Suppression d'un acquittement d'une opération	104
5.3	LTS du médiateur pour la suppression d'un choix d'opération	105
5.4	Méthode de résolution de suppression d'un terme d'une alternative	106
5.5	LTS du médiateur pour la suppression d'un acquittement	106
5.6	Méthode de résolution de suppression d'acquiescement	107
5.7	Algorithme de résolution des incompatibilités	108
5.8	Interactions via des médiateurs	109
5.9	Génération de médiateurs pour plusieurs interfaces.	111
6.1	Architecture du canevas <i>ArchiMed</i>	119
6.2	Diagramme des paquetages du canevas <i>ArchiMed</i>	119
6.3	Représentation des automates dans le format SCXML	120
6.4	Diagramme des classes de la détection	122
6.5	<i>ArchiMed</i> : vue globale du canevas	123
6.6	Détection des incompatibilités	124
6.7	Résultats sous forme textuelle de la détection des incompatibilités	125
6.8	Résultats sous forme graphique de la détection des incompatibilités	126
6.9	Résultats textuels de la mesure de similarité	127
6.10	Détection des incompatibilités résolubles	128
6.11	Résultats graphiques de la mesure de similarité	129
6.12	Résultats des comparaisons des mesures de similarité	133

tel-00351255, version 1 - 8 Jan 2009

Première partie

Contexte de la recherche et état de l'art.

Chapitre 1

Contexte des services web et problématique

Sommaire

1.1	Introduction	5
1.2	Concepts fondamentaux	8
1.2.1	Communication par envois et réceptions de messages	8
1.2.2	Interfaces des services web	9
1.2.3	Composition de services web : orchestration et chorégraphie	12
1.2.4	Conformité et équivalence des interfaces	14
1.3	Problématique : incompatibilités des interfaces	17
1.3.1	Comportement et structure	18
1.3.2	Ajout d'une opération	19
1.3.3	Suppression d'une opération	20
1.3.4	Modification des opérations dans une séquence	21
1.4	Objectifs de la thèse et approche ArchiMed	22
1.5	Plan de la thèse	23

1.1 Introduction

Les architectures distribuées s'appuient de plus en plus sur des technologies et des standards de l'Internet afin de permettre aux applications d'interagir les unes avec les autres, et ce de manière faiblement couplée. Les services web figurent parmi ces technologies. Un service web peut être vu comme une application qui offre des fonctionnalités

accessibles via le web par d'autres applications qui requièrent ces fonctionnalités. Un service web est un fournisseur d'opérations. Dans la suite de la thèse, les termes *service web*, *service* et *fournisseur*, sont employés avec le même sens. Les applications qui utilisent les opérations offertes par un fournisseur sont dites applications clientes. Ainsi, dans la suite de la thèse, le terme *client* désigne l'application cliente qui utilise les opérations d'un fournisseur. La mise en œuvre de services web s'appuie sur l'utilisation de plusieurs standards basés sur XML pour la description des caractéristiques fonctionnelles et non fonctionnelles de ces services (WSDL : *Web Service Description Language* [55], BPEL : *Business Process Execution Language* [44], WSCI : *Web Service Choreography Interface* [54], etc.), et sur un standard pour la définition des formats des messages échangés entre les services (SOAP : *Simple Object Access Protocol* [53]). Les interactions entre les services web se font par envois de messages. REST (*REpresentational State Transfer*) est une autre approche proposée pour exploiter des services web qui exposent des ressources aux clients [31].

Dans les technologies à base de composants distribués, le couplage entre les objets est fort. A titre d'exemple, RMI (*Remote Method Invocation*) est une API (*Application Program Interface*) Java permettant de manipuler des objets distants. Les communications s'effectuent grâce au protocole RMI-IIOP (*Internet Inter-Orb Protocol*), un protocole normalisé par l'OMG (*Object Management Group*) et utilisé dans l'architecture CORBA. La norme CORBA de l'OMG permet de manipuler des objets à distance avec n'importe quel langage. La mise en œuvre de CORBA nécessite une connaissance de la structure des objets manipulés par les applications clientes et par les applications fournisseurs. Des règles de transformation objets/messages doivent être définies au préalable par les partenaires.

Dans le cas des services web, les interactions sont basées sur des envois et des réceptions de messages. Les opérations offertes par un service sont propriétaires et elles ne sont accessibles que via une interface. Une composition de services est constituée de plusieurs services web qui interagissent les uns avec les autres afin d'offrir de nouvelles fonctionnalités qui créent une plus-value. La conception de telles compositions implique une collaboration entre les différents services dits partenaires. Mais aussi, la cohérence dans les schémas d'interactions doit être garantie et vérifiée. Après la mise en œuvre du service composite, le suivi des interactions et des évolutions des services partenaires est assuré par une activité de contrôle [25]. En cas d'incohérence, la conception de la composition et la définition des différents liens d'interactions qui induisent des incohérences sont revues et corrigées.

La mise en œuvre d'une application cliente, est réalisée par un concepteur sur la base des opérations fournies par les services qu'il prévoit d'utiliser. Dans ce cas, le concep-

teur est supposé avoir accès à la description de l'interface des services et aux adresses où chacun est accessible. Cependant, si l'application cliente requiert d'autres opérations, et que le concepteur ne connaît pas quel service offre ces opérations, alors le concepteur doit découvrir dans le web les services qui offrent ces opérations. Afin de faciliter la découverte de services, des annuaires de services sont proposés. Un annuaire de services permet à des services web de s'enregistrer en fournissant les informations nécessaires pour les localiser sur le web et les utiliser. Une fois que les services sont inscrits, les clients peuvent interroger l'annuaire pour découvrir le service qui répond au mieux à leurs besoins. UDDI (*Universal Description Discovery and Integration*) [47, 21] est un standard pour la définition de tels annuaires. Un annuaire de services UDDI permet aux services web de s'inscrire en fournissant, entre autres, leurs URI (*Uniform Resource Identifier*) et leurs descriptions d'interfaces dans le standard *WSDL* afin qu'ils soient visibles aux autres services qui veulent les utiliser. Cependant, l'annuaire ne reflète pas toujours la disponibilité des services qui y sont inscrits.

Les systèmes d'information des entreprises doivent être souples et extensibles pour intégrer des modifications et des changements imposés par des besoins internes ou par des contraintes externes [29, 70, 69]. Ces évolutions sont parfois synonymes d'incompatibilités entre le système d'information de l'entreprise qui évolue et ceux de ses différents partenaires. Ces incompatibilités doivent être résolues par les partenaires. Ces derniers définissent de nouvelles applications qui utilisent les opérations offertes dans la nouvelle interface du système. La détection et la résolution manuelles des incompatibilités ont un coût qui peut pousser un partenaire à chercher d'autres alliances. La recherche d'une nouvelle alliance est réalisée par la découverte d'un nouveau fournisseur dont l'interface fournie est compatible avec l'interface requise du partenaire.

Dans le contexte des services web, l'évolution des interfaces est étudiée sous deux angles différents mais complémentaires : structurel et comportemental. L'interface structurelle aussi bien que l'interface comportementale d'un service web peuvent être modifiées par leur propriétaire. Ces changements dans l'interface du service ne sont pas pris en compte dans les applications clientes qui utilisent le service. Des modifications dans l'interface du service induisent souvent des incompatibilités avec les interfaces requises par les clients. Etant donné que les applications clientes sont propriétaires, c'est au concepteur de ces dernières de prendre en compte les changements de l'interface du service. Le concepteur doit mettre en œuvre de nouvelles applications qui utilisent les opérations de la nouvelle interface du service. C'est dans cette problématique de changement des interfaces des services que s'inscrit la thèse présentée dans ce document.

Dans la suite de ce chapitre, les concepts fondamentaux des services web sont introduits dans la section 1.2. La section 1.3, présente la problématique liée aux incompatibilités comportementales au cours des interactions entre des services web suite aux changements de leurs interfaces. Puis, dans la section 1.3.1 est détaillée le problème lié à l'incidence de la modification de la structure d'un service sur son comportement. Les trois sections 1.3.2, 1.3.3 et 1.3.4 exposent les cas élémentaires de changement de comportement (ajout, suppression et modification d'opérations) qui provoquent des incompatibilités.

1.2 Concepts fondamentaux

Dans cette section, nous présentons les différents concepts manipulés dans le contexte des services web. Le principe de communication des services est introduit dans la section 1.2.1. La définition des interfaces est présentée dans la section 1.2.2. Dans la section 1.2.3, deux points de vue de la conception de compositions de services web sont abordés : l'orchestration et la chorégraphie. Les définitions de la conformité et de l'équivalence des interfaces sont présentées dans la section 1.2.4.

1.2.1 Communication par envois et réceptions de messages

La figure 1.1 schématise une situation où un service fournisseur *Entrepôt* expose ses fonctionnalités (mises en œuvre dans un langage donné) par le biais d'une interface décrite en WSDL (voir par exemple, les opérations *commander*, *payer* et *livrer*). Les mécanismes de communication des services web sont basés sur l'envoi et la réception de messages textuels via un réseau. Le format de ces messages est fixé par un standard tel que SOAP qui est basé sur XML. Les messages SOAP sont transportés sur le réseau par le biais d'autres protocoles standards de communication tels que HTTP, FTP, etc. Un message SOAP est construit par un service (par exemple, *Magasin*) dans la perspective d'être envoyé à destination d'un autre service (par exemple, *Entrepôt*). Le message contient, entre autres, les détails sur l'opération que *Magasin* demande à *Entrepôt* d'exécuter. Le message contient aussi les valeurs des paramètres nécessaires à l'exécution de cette opération. Dans le cas d'opérations *synchrones* (de type *RPC*), le message peut ne contenir que la valeur de retour de cette opération. Dans les services web, les opérations synchrones sont des opérations où chaque envoi de message est systématiquement suivi d'une réception de message [4]. C'est un mécanisme qui reprend l'idée d'appel à une opération distante [43]. Si l'opération est *asynchrone*, alors elle prend en paramètre un message en entrée (ou message en réception) uniquement [4].

Les enchaînements entre les opérations fournies sont décrits dans autre standard tel que BPEL (voir par exemple, l'opération *commander* suivie de l'opération *payer* et enfin l'opération *livrer*). Les interactions entre un client **Magasin** et le fournisseur **Entrepôt** sont initiées par le client par l'envoi d'un message contenant les détails nécessaires à l'exécution de l'opération *commander*. Dans la séquence de messages décrite dans la figure 1.1, le service client envoie un message pour l'exécution de l'opération synchrone de commande avec en paramètre la liste des articles commandés (*commander(ListArt)*). L'opération de commande retourne en résultat un message de confirmation (*réponseCmd*) de réception de la commande. Puis le service client effectue le paiement de la commande en envoyant le message *payer(DétailsCC)*. Ce message porte les détails de la carte de crédit nécessaires à l'exécution de l'opération de paiement (*payer(détailsCC)*). Un dernier message, de notification de la livraison (*livrer(dateLiv)*) est envoyé par le service fournisseur au client.

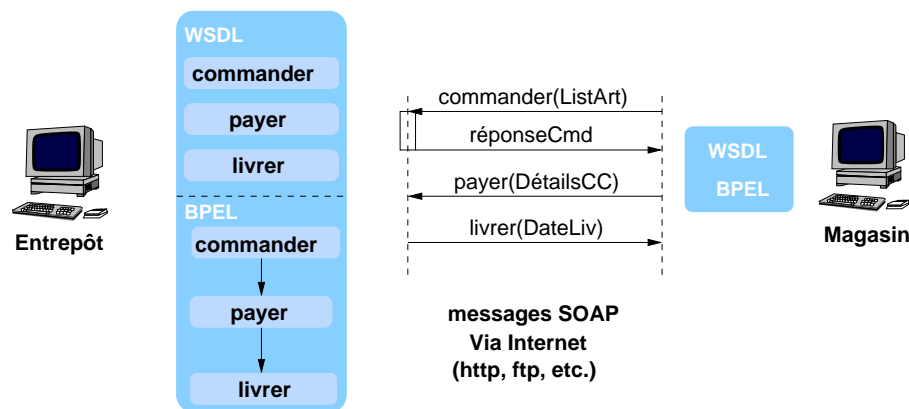


FIG. 1.1 – Communication entre des services web

1.2.2 Interfaces des services web

Un service fourni par un composant logiciel est défini par une *interface*, qui est une description concrète de l'interaction entre le client et le fournisseur du service [72]. L'auteur définit deux vues complémentaires d'une interface.

- la vue d'usage : une interface définit les opérations et les structures de données utilisées pour la fourniture d'un service, c'est l'interface *fournie* ;
- la vue contractuelle : une interface définit un contrat entre le client et le fournisseur d'un service, c'est l'interface *requise*.

L'interface d'un service web comporte également la description des caractéristiques non fonctionnelles du service (par exemple, les caractéristiques de qualité de service) [16]¹. Le contrat associé à l'interface spécifie par exemple des contraintes sur l'ordre d'exécution des opérations de l'interface.

Pour illustrer notre propos, nous considérons une nouvelle fois le service web Entrepôt. Les clients peuvent interagir avec le fournisseur par le biais d'opérations de commande, de paiement et de livraison exposées dans son interface. La figure 1.2 illustre les différentes étapes du processus de vente exposées dans l'interface du fournisseur. Le service Entrepôt attend une commande d'un client (voir l'activité Réception commande). Une fois cette commande reçue, le fournisseur envoie une confirmation au client qui a initié la conversation (voir Envoi confirmation). Le fournisseur offre la possibilité au client de modifier ou d'annuler sa commande (voir Réception modification et Réception annulation). Si le fournisseur reçoit le paiement de la commande (voir Réception paiement), alors les articles commandés seront livrés au client (voir Envoi livraison). Dans le cas où le fournisseur reçoit un message d'annulation de la commande, la conversation avec le client se termine sans aboutir au paiement ni à la livraison de la commande.

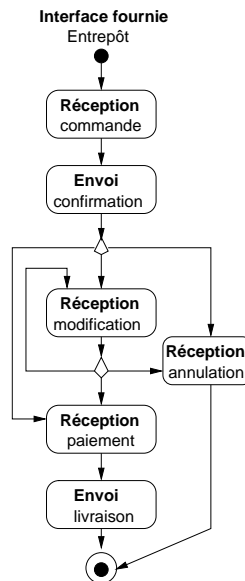


FIG. 1.2 – Interface fournie du service Entrepôt (notation UML : diagramme d'activités)

D'après cet exemple, afin qu'un client puisse effectuer des achats d'articles ou de services auprès du fournisseur Entrepôt, ce dernier doit exposer les signatures des opérations de commande de paiement et de livraison, dans son *interface structurelle* et doit égale-

¹L'étude des caractéristiques non-fonctionnelles d'un service est en dehors du cadre de cette thèse.

ment exposer le processus métier qui décrit les enchaînements de ces opérations dans son *interface comportementale*.

L'*interface structurelle* d'un service permet de décrire les signatures des opérations fournies par le service (les noms des opérations, les paramètres des opérations, le types des paramètres, etc). Le standard *WSDL* est utilisé pour décrire cette interface (voir l'exemple donné en annexe B). L'interface structurelle du service *Entrepôt* contient les signatures des opérations de commande, de paiement, de livraison, de modification et d'annulation d'une commande, avec les structures des messages envoyés et reçus.

L'*interface comportementale* d'un service permet de décrire les enchaînements des opérations décrites dans l'interface structurelle. L'interface comportementale peut être décrite dans des standards tels que *BPEL4WS*² (*Business Process Execution Language for Web Services*) ou *WSCI* (*Web Service Choreography Interface*) [116]. Ces langages offrent les opérations classiques des langages de programmation pour décrire des enchaînements d'interactions (séquence, parallélisme, alternative, itération, etc.). Par exemple, dans l'interface comportementale du service *Entrepôt*, l'enchaînement est décrit dans l'interface comportementale comme suit. Les opérations pour la réception d'une commande et pour l'envoi d'une confirmation au client sont décrites dans une séquence qui spécifie l'ordre d'exécution de ces opérations. Après l'envoi de la confirmation par *Entrepôt* à destination du client, ce dernier a trois choix d'opérations (décrits dans une alternative) qui sont l'annulation, le paiement ou la modification de la commande. Si le client choisit de modifier la commande, alors il aura de nouveau les trois choix d'opérations. Le client peut ainsi modifier la commande autant de fois qu'il le souhaite (ce choix est décrit par une itération). La dernière séquence d'opérations indique qu'après le paiement, la commande sera livrée au client.

Les conversations entre les services : les services web interagissent les uns avec les autres par le biais d'envois de messages. Dans ce cadre, une interaction entre deux services, un client et un fournisseur, est décrite par un envoi de message par le client et par la réception de ce message par le fournisseur. De cette interaction peuvent découler d'autres interactions, au cours desquelles les rôles de client et de fournisseurs peuvent s'inverser. Une séquence de telles interactions est appelée une conversation [72].

Dans l'exemple de la figure 1.3, l'interface requise du client *Magasin* décrit l'enchaînement des envois et des réceptions de messages qui correspondent à l'enchaînement tel qu'il est spécifié dans l'interface fournie du service *Entrepôt*. Par exemple, dans l'interface requise de *Magasin*, l'opération d'envoi du message de commande est attendue par

²Devenu plus tard BPEL.

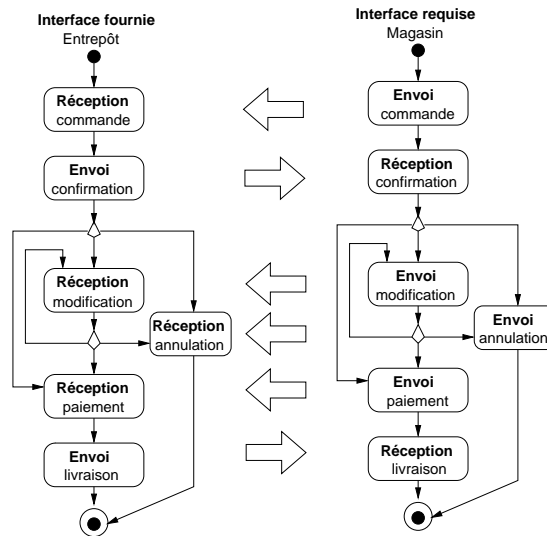


FIG. 1.3 – Interface fournie versus requise

une opération de réception de ce message dans l'interface fournie de Entrepôt. Ce dernier envoie un message de confirmation à destination du client. Dans son interface fournie, le service Entrepôt offre à son client la possibilité de modification ou d'annulation de la commande. Enfin le client Magasin envoie un message de paiement qui est attendu par l'opération de réception de paiement, puis le client attend la réception du message de livraison que le fournisseur Entrepôt envoie.

1.2.3 Composition de services web : orchestration et chorégraphie

Un service dit composite est constitué de plusieurs services qui interagissent les uns avec les autres. Cette composition de services offre de nouvelles opérations, décrites dans son interface structurelle, et utilise les opérations des services composants. L'interface comportementale du service composite spécifie un nouvel enchaînement des opérations. Une composition de services peut être considérée selon deux différents points de vue de la conception, la *chorégraphie* et l'*orchestration* [10, 90, 98].

Une *chorégraphie* modélise, d'une part un ensemble d'interactions qui peuvent ou doivent avoir lieu entre un ensemble de services (représentés de façon abstraite par des rôles), et d'autre part les dépendances entre ces interactions [72]. Au moment de la conception d'une chorégraphie, les rôles de la composition sont spécifiés. Un rôle, doit fournir certaines opérations. Dans un rôle, des liens décrivent les échanges des messages avec les autres rôles de la chorégraphie [90]. Ainsi, chaque rôle possède une description partielle

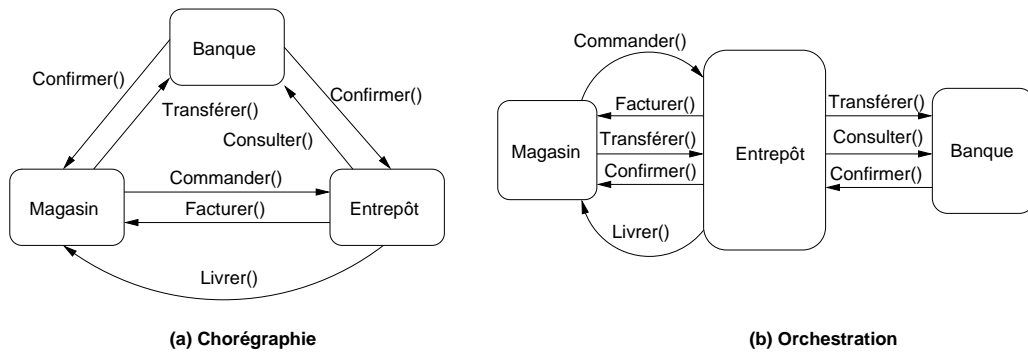


FIG. 1.4 – Chorégraphie et orchestration de services web

de la composition. Le service doit aussi interagir avec les autres services selon l'enchaînement des échanges de messages décrit dans son rôle. L'exemple de la figure 1.4 (a) illustre la modélisation d'une composition de services web selon une chorégraphie (voir figure 1.4 (a)). A l'exécution de la chorégraphie, le service web **Magasin** interagit avec les deux services **Banque** et **Entrepôt**, selon sa description de l'enchaînement des opérations. Cependant, le service **Magasin** n'a pas la description des interactions entre les services **Banque** et **Entrepôt**. En d'autres termes, les échanges de messages qui porte sur les opérations de consultation et de confirmation entre la banque et l'entrepôt ne sont pas définies dans l'interface du service magasin.

Une orchestration modélise un ensemble d'actions communicationnelles et d'actions internes dans lesquelles un service donné peut ou doit s'engager (afin de remplir ses fonctions) ainsi que les dépendances entre ces actions. Une orchestration peut être vue comme le raffinement d'une interface qui inclut des actions qui ne sont pas nécessairement pertinentes pour les clients du service mais qui servent à remplir les fonctions que le service fournit et doivent donc être prises en compte dans son implantation [72]. L'orchestration décrit l'enchaînement des échanges des messages entre le composite est tous les autres partenaires de l'orchestration. A l'exécution de l'orchestration, les services partenaires sont sollicités par le service composite qui possède une description globale des interactions. Dans l'exemple de la figure 1.4 (b), l'orchestration décrit la composition de trois services **Magasin**, **Banque** et **Entrepôt**. Les seules interactions modélisées sont les interactions entre le service composite **Entrepôt** et chacun des services partenaires (le magasin et la banque).

1.2.4 Conformité et équivalence des interfaces

Dans cette section, nous discutons de la notion de conformité d'interfaces des services, qui lorsqu'elle est démontrée assure que les conversations entre les services peuvent se dérouler selon le processus tel qu'il est prévu par le fournisseur.

Une interface requise $I2$ est dite conforme à une interface fournie $I1$ si l'interface $I2$ et $I1$ vérifient la conformité structurelle et la conformité comportementale.

La conformité structurelle : une interface requise $I2$ est dite conforme à une interface fournie $I1$ sur le plan structurel si et seulement si la structure de chaque message envoyé (respectivement reçu) décrit dans l'interface fournie est la même que la structure du message reçu (respectivement envoyé) décrit dans l'interface requise. Deux structures de messages sont identiques si elles ont les mêmes signatures d'opérations et les mêmes définitions des paramètres et de leurs types.

La conformité comportementale : une interface requise $I2$ est dite conforme à une interface fournie $I1$ sur le plan comportemental si et seulement si l'enchaînement des opérations d'envoi (respectivement de réception) décrit dans l'interface requise correspond à l'enchaînement des opérations de réception (respectivement d'envoi) décrit dans l'interface fournie [13].

Une conversation cohérente entre un client et un fournisseur est un ensemble d'échanges de messages qui s'exécutent selon l'enchaînement des interactions décrites dans l'interface comportementale du fournisseur [13]. A la conception, et dans la perspective de garantir la cohérence de la conversation entre le fournisseur et le client, la conformité structurelle et comportementale entre l'interface fournie par le service et celle requise par le client doit être établie.

L'exemple de la figure 1.5 illustre le cas où l'interface fournie par le service Entrepôt est conforme à celle requise par le client Magasin1). Cette figure 1.5 illustre également et le cas où l'interface fournie par le service Entrepôt n'est pas conforme à celle requise par le client Magasin2). La conformité structurelle entre l'interface fournie par le service Entrepôt et l'interface requise du client Magasin1 est vérifiée car toutes les structures des messages envoyés et reçus sont identiques (voir, commande, confirmation, facture, transfert et livraison)³. A chaque envoi (respectivement de réception) de message dans l'interface fournie lui correspond une réception (respectivement d'envoi) de message dans l'interface requise. En effet, l'envoi du message commande par le client Magasin1, qui initie la conversation, est attendu par la réception du message commande dans l'interface du fournisseur

³Nous prenons l'hypothèse que les messages de même nom désignent des messages qui ont la même structure.

Entrepôt. Puis l'envoi du message confirmation est déclenché par le fournisseur Entrepôt et à destination du client Magasin1 qui reçoit ce message confirmation. Les enchaînements des envois et des réceptions de messages se poursuivent de manière conforme pour les phases de facturation et de paiement, jusqu'à la réception de la livraison qui clôt la conversation. Ainsi, l'interface requise de magasin1 est conforme sur le plan comportemental à l'interface fournie par Entrepôt.

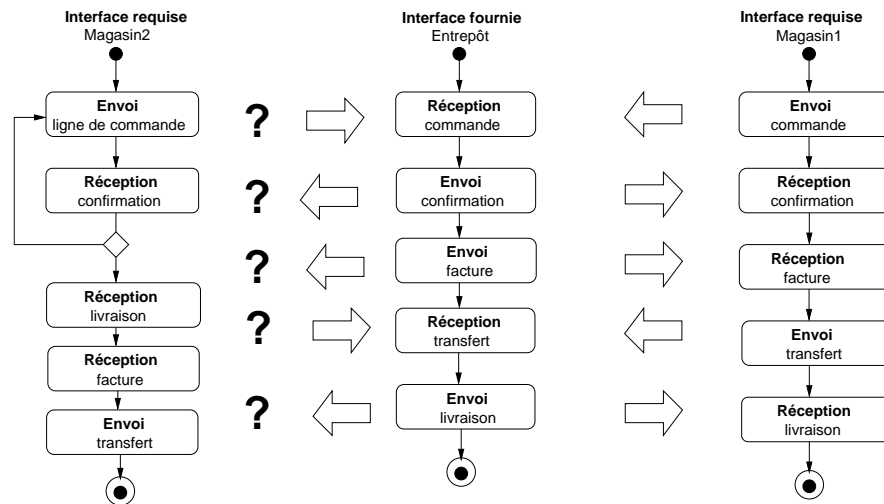


FIG. 1.5 – Conformité des interfaces fournie et requise

Dans la conversation entre le client Magasin2 et le service Entrepôt, décrite dans la figure 1.5, la conformité n'est pas vérifiée, ni sur le plan structurel, ni sur le plan comportemental. En effet, au niveau de la vérification de la conformité structurelle, la structure du message ligne de commande requise par le client Magasin2 n'est pas fournie par l'interface du fournisseur (voir le message commande). Le fournisseur peut recevoir un seul message de commande alors que le client Magasin2 envoie plusieurs lignes de commande. La répétition des envois de confirmation à la réception de chaque ligne de commande envoyé par Magasin2 n'est pas prévue dans l'interface fournie. Cette notion est définie dans [29] comme *envois multiples versus réception unique*. En d'autres termes, le client effectue plusieurs envois de messages (*envois multiples*) dont les structures sont identiques et que le fournisseur attend la réception d'un seul message (*réception unique*) dont la structure est composée de plusieurs sous-structures. Dans les étapes de facturation et de paiement de la commande, le fournisseur envoie la facture au client Magasin2 qui ne prévoit pas de recevoir ce message. En effet, le client attend le message de livraison de la commande que le fournisseur ne lui envoie pas (l'envoi de la facture ne correspond pas à la réception de celle-ci). Ainsi, sur le plan comportemental, l'interface fournie du fournisseur Entrepôt n'est pas conforme à l'interface requise du client Magasin2.

Équivalence contextuelle et globale des interfaces fournies : une équivalence contextuelle ou partielle est définie pour deux interfaces fournies $I1$ et $I2$ par rapport à une interface requise R si et seulement si R est conforme avec $I1$ et R est également conforme avec $I2$. $I1$ et $I2$ sont équivalentes par rapport à un contexte défini par une interface requise R . Les deux interfaces fournies $I1$ et $I2$ peuvent se substituer mutuellement dans une conversation qui implique l'interface requise R d'un client [13]. En effet, un client dont l'interface requise est R qui interagit avec un service dont l'interface fournie est $I1$ peut substituer ce service par un autre service dans l'interface fournie est $I2$ et inversement.

L'équivalence globale (ou indépendante du contexte) de deux interfaces fournies $I1$ et $I2$ est vérifiée si et seulement si, quelque soit l'interface requise, les deux interfaces fournies peuvent se substituer mutuellement. Dans [20], l'équivalence entre deux interfaces est définie comme une bisimulation des représentations en automates de ces deux interfaces. La bisimulation entre deux interfaces désigne le fait que chaque interface simule le comportement de l'autre. Une première interface simule une deuxième interface, si la première interface inclut le comportement de la deuxième interface.

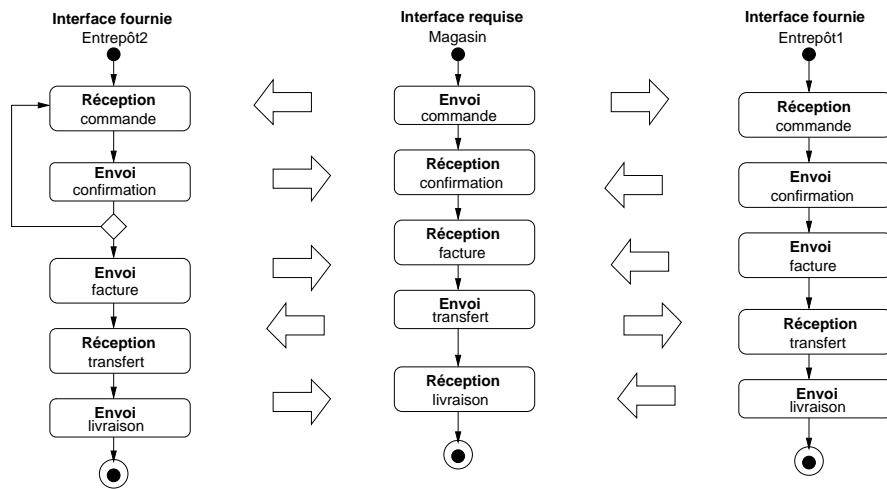


FIG. 1.6 – Équivalence contextuelle des interfaces fournies

Dans l'exemple de la figure 1.6, l'interface fournie du fournisseur Entrepôt1 décrit un processus en attente d'un message décrivant la prise en compte d'une commande (voir Réception commande) envoyé par le client Magasin (voir Envoi commande). Le fournisseur Entrepôt1 envoie une confirmation de commande suivie d'un message de facturation (voir Envoi confirmation et Envoi facture) au magasin. Après avoir reçu ces deux messages, le client effectue le paiement de la commande par l'envoi du message de transfert (voir Envoi transfert). Après la réception du message de transfert, Entrepôt1 envoie le message de

livraison qui est attendu dans l'interface requise du client **Magasin**. Dans l'interface fournie du service **Entrepôt2**, le processus est identique à celui de l'interface du service **Entrepôt1** sauf que **Entrepôt2** offre à son client **Magasin** la possibilité de modifier sa commande (voir le flux qui boucle sur la réception de commande après l'envoi d'une confirmation). L'interface requise du service client est conforme avec les deux interfaces fournies de **Entrepôt1** et **Entrepôt1**. En effet, comme décrit dans son interface requise, lorsque le client envoie une commande il ne la modifie plus. Ainsi, les deux interfaces fournies, des services **Entrepôt1** et **Entrepôt2**, sont partiellement équivalentes selon le contexte défini par l'interface requise du client **Magasin**.

1.3 Problématique : incompatibilités des interfaces

L'étude des incompatibilités entre deux interfaces $I1$ et $I2$ est justifiée à plusieurs titres. Lorsque $I2$ est issue d'une évolution de $I1$, étudier les incompatibilités entre $I2$ et $I1$ permet de mettre en évidence les modifications qui ont été effectuées. L'objectif de cette étude est de propager sur les clients les adaptations nécessaires pour qu'ils continuent à communiquer avec le service dans son nouvel état. Dans certains cas, cette adaptation peut être évitée par la fourniture d'un médiateur qui réconcilie les conversations entre les clients et le service. Dans le contexte de la gestion de processus métiers les modèles de processus métiers décrivent les enchaînements des activités et des interactions entre les différents partenaires. A l'exécution du processus métier, des instances du modèle du processus sont créées et permettent de rendre compte de l'état d'exécution des activités. La définition d'un nouveau modèle de processus métiers doit tenir compte des différences qui existent avec l'autre modèle du processus. Ces différences engendrent des incompatibilités dans les interactions entre les partenaires. Les incompatibilités dans les instances de processus sont en cours d'exécution doivent être étudiées [114]. Les modifications nécessaires sont propagées aux différents partenaires pour éviter les incompatibilités [71]. Dans cette perspective, des métriques sont proposées pour mesurer le degré de différence entre deux processus métiers [74]. Les modifications des interfaces, en génie logiciel, sont considérées comme des optimisations des programmes et/ou comme des extensions des codes par de nouvelles fonctionnalités [24]. Dans le contexte des services web, les seules opérations modélisées sont les opérations communicationnelles d'envoi et de réception de messages. Les services web sont considérés comme des boîtes noires où les modifications des opérations internes qui n'ont pas d'impact sur les définitions des opérations de communication ne sont pas prises en compte.

A travers un scénario exemple, nous explicitons le problème des incompatibilités générées entre les interfaces fournie et requise lorsque la définition de l'interface fournie

d'un service change. Dans notre étude l'accent est mis sur les incompatibilités comportementales qui ne sont pas totalement disjointes des incompatibilités structurelles ainsi que nous le détaillons dans la section 1.3.1. La définition de l'interface comportementale d'un service change lorsqu'une opération est ajoutée dans une séquence. Un exemple d'incompatibilité générée suite à ce type de changement est présenté dans la section 1.3.2. La suppression d'une opération dans une séquence produit des incompatibilités comme présenté dans la section 1.3.3. Un autre exemple d'incompatibilité due à la modification des opérations dans une séquence est présenté dans la section 1.3.4.

1.3.1 Comportement et structure

La modification de l'interface fournie d'un service est rendue nécessaire lorsque la définition de la structure des messages échangées évolue. Nous parlons alors d'un changement structurel de l'interface fournie. Dans certains cas, un changement structurel de l'interface est accompagné d'un changement dans l'interface comportementale du service. Un changement dans l'interface comportementale se produit lorsque l'enchaînement des envois et des réceptions des messages décrits dans l'interface fournie a changé. Les changements introduits dans une interface $I1$ entraînent la définition d'une nouvelle interface $I2$.

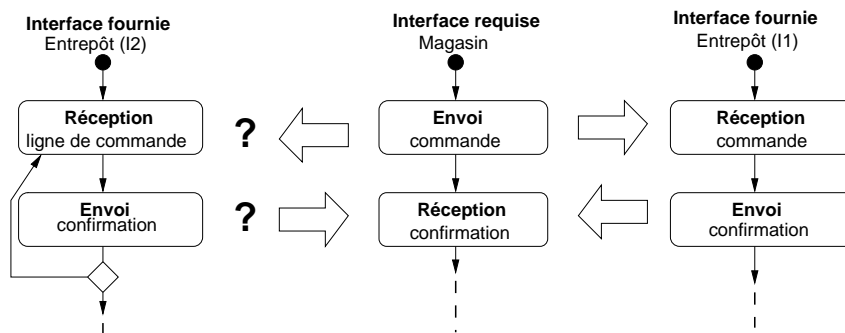


FIG. 1.7 – Changement de la structure des messages

Dans l'exemple de la figure 1.7, l'interface fournie du service Entrepôt dans sa définition I1, décrit la réception d'un message commande envoyé par un client Magasin. La structure du message envoyé regroupe toutes les lignes d'une commande. L'interface requise du client est conforme à cette interface fournie I1. Après la définition d'une nouvelle interface I2 du service fournisseur, les lignes de commande d'une commande doivent être envoyées séparément et un message de confirmation doit être retourné vers le client pour chaque ligne de commande. La conformité entre l'interface fournie I2 et l'interface requise par le client Magasin n'est plus vérifiée. Dans cet exemple, la structure du message commande, envoyée par le client en un seul message, a changé. En effet, la structure du

message commande est éclatée en plusieurs messages qui représentent chacun une ligne de la commande. Ce changement dans la structure du message de commande entraîne un changement dans le mécanisme d'envoi de la commande qui est remplacé par plusieurs envois des différentes lignes de la commande. Ce changement structurel de l'interface fournie du service a engendré une incompatibilité comportementale entre l'interface fournie et l'interface requise car le client ne peut pas envoyer plusieurs lignes de commande.

1.3.2 Ajout d'une opération

Des changements dans l'interface d'un service peuvent être opérés par l'ajout d'une ou de plusieurs opérations dans une séquence d'opérations. Ces nouvelles opérations doivent être prises en compte dans l'interface requise du client. Dans le cas contraire, des incompatibilités vont survenir car le client n'a pas défini dans son interface requise les mécanismes d'échanges de messages avec cette nouvelle opération. Les clients qui continuent d'interagir avec le service selon la nouvelle définition de l'interface fournie seront confrontés à des incompatibilités comportementales qui font échouer les conversations qu'ils ont engagées avec le service.

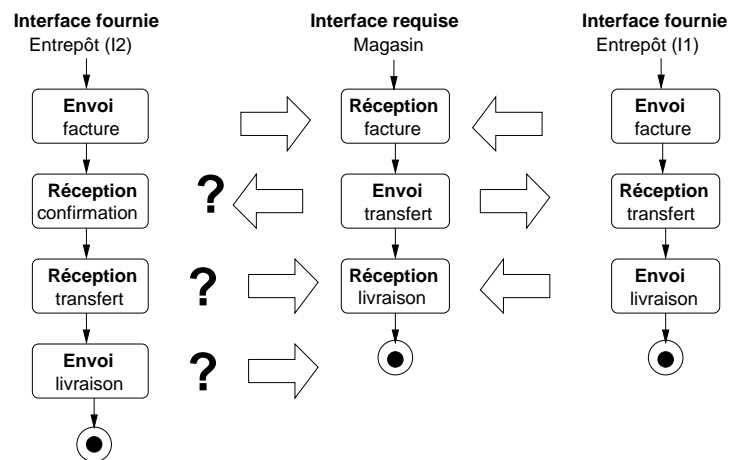


FIG. 1.8 – Ajout d'une opération

Dans l'exemple de la figure 1.8, l'interface fournie du service Entrepôt, dans sa définition I1, n'exige pas de réception de confirmation après l'envoi de la facture au client Magasin. Arrivée à l'étape de facturation de la commande, la conversation entre le client et le fournisseur se déroule comme suit (voir le schéma de conversation entre le client Magasin et le fournisseur Entrepôt (I1)) : le fournisseur envoie le message *facture* qui est reçu par le client. Ce dernier envoie le message *transfert* qui est reçu par le fournisseur. Enfin, le fournisseur envoie le message *livraison* qui est reçu par le client. Cette conversation

se termine avec succès. Dans la nouvelle définition de l'interface fournie l2, une opération de réception de confirmation de transfert est ajoutée (voir le schéma de conversation entre le client Magasin et le fournisseur Entrepôt (l2)). En effet, dans la nouvelle définition de l'interface fournie, le fournisseur attend la réception du message confirmation suite à l'opération d'envoi de la facture. Cependant, le client Magasin enchaîne la conversation sur l'envoi du message transfert. Cette conversation n'aboutit pas car le fournisseur Entrepôt (l2) attend indéfiniment la réception du message confirmation que le client n'enverra jamais.

1.3.3 Suppression d'une opération

La suppression d'une opération dans une séquence décrite dans une interface fournie est un changement qui induit des incompatibilités avec les interfaces requises des clients. Ces derniers continuent d'interagir avec le service selon la définition de l'interface fournie avant qu'elle ne change et sollicitent l'opération supprimée or, cette opération n'existe plus dans la nouvelle description de l'interface. Si l'opération supprimée de l'interface est un envoi de message, alors le service client va attendre indéfiniment ce message qui ne va pas arriver. Dans le cas où l'opération supprimée est une réception de message, le client envoie un message qui ne sera jamais accepté (reçu) par le fournisseur, engendrant ainsi une incompatibilité dans la conversation.

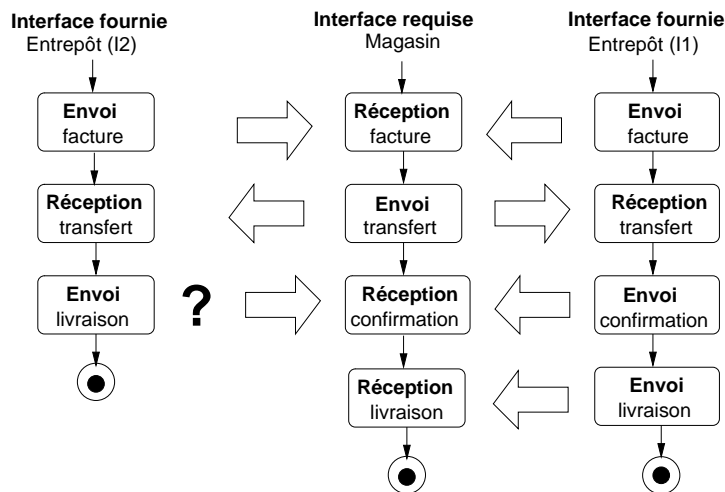


FIG. 1.9 – Suppression d'une opération

L'exemple de la figure 1.9 illustre les interactions entre un client Magasin (selon sont interface requise) et un fournisseur Entrepôt (selon ses deux définitions de l'interface fournie (l1) et (l2)). La définition de l'interface (l1) du fournisseur décrit une séquence d'opé-

rations où la réception d'un message de transfert est suivie d'une opération d'envoi de confirmation (voir les interactions entre le client **Magasin** et le fournisseur **Entrepôt (I1)**). Le client attend la réception du message de confirmation avant de recevoir le message de la livraison. La définition de l'interface fournie change, suite à la suppression de l'opération de confirmation de transfert, vers une nouvelle définition (I2). L'interface fournie dans sa nouvelle définition prend pour hypothèse dans sa logique métier que désormais l'envoi de la livraison peut être interprété comme une confirmation à la réception du transfert. Cependant, le client qui continue d'interagir avec ce service selon (I1) va attendre indéfiniment le message de confirmation du transfert qui ne va jamais arriver (voir les interactions entre le client **Magasin** et le fournisseur **Entrepôt (I2)**). Dans ce cas, l'envoi du message de livraison par **entrepôt(I2)** échoue car le client attend la réception de la confirmation.

1.3.4 Modification des opérations dans une séquence

Lorsqu'une opération est modifiée par une autre opération dans l'enchaînement d'opérations décrit dans l'interface d'un service, les clients ne peuvent plus interagir avec le service. L'opération attendue par les clients a été modifiée par une autre opération dans l'interface fournie. Cette incompatibilité empêche la conversation entre le service et le client de se terminer.

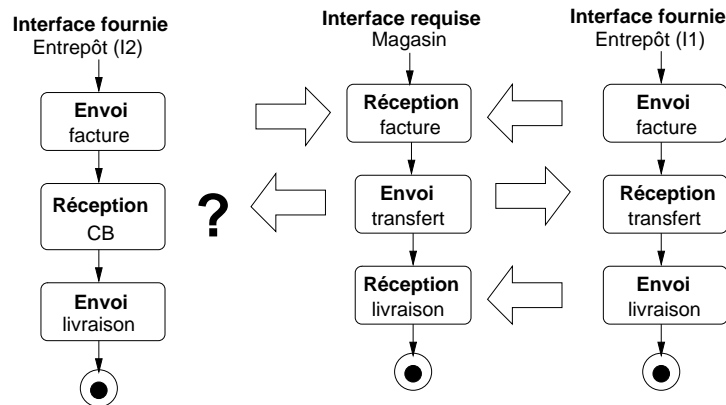


FIG. 1.10 – Modification d'une opération

Dans l'exemple de la figure 1.10, l'enchaînement des opérations décrit dans l'interface fournie du service **Entrepôt (I1)** est comme suit (voir les interactions entre le client **Magasin** et le fournisseur **Entrepôt (I1)**) : le client reçoit le message **facture**, puis le client envoie le message **transfert** pour le paiement de la facture. Enfin le fournisseur envoie le message **livraison** à destination du client pour terminer la conversation avec succès. Dans la nouvelle définition I2 de l'interface fournie du service, l'opération de réception du message **transfert**

est modifiée en une opération de réception du message de Coordonnées Bancaires (CB). De ce fait, cette conversation n'aboutit pas car le client Magasin envoie un message de transfert qui ne sera jamais reçu par le fournisseur Entrepôt (I2). Ce dernier attend un message des coordonnées bancaires qui ne sera pas envoyé par le client. D'un autre côté, le client attend la réception du message de livraison qui n'arrivera pas.

1.4 Objectifs de la thèse et approche ArchiMed

La thèse rapportée dans ce document étudie la détection et la de résolution des incompatibilités engendrées par des différences dans les définitions des interfaces comportementales des services web. Pour ce faire, nous proposons une approche que nous explicitons brièvement avant de la détailler dans la partie proposition. Dans un premier temps, une modélisation des interfaces comportementales par des automates est adoptée. Dans cette modélisation, seul le comportement externe (comportement observable) est considéré. En d'autres termes, il n'y a que les opérations d'envoi et de réception de messages qui sont décrites dans les interfaces. Une fois les interfaces comportementales décrites en automates, une étape de détection des incompatibilités entre les différentes définitions de l'interface d'un service est réalisée. La détection des incompatibilités est automatique et elle est exécutée en deux étapes : *la détection des incompatibilités élémentaires* et *la détection des incompatibilités automatiquement résolubles*. Les incompatibilités élémentaires sont induites par des changements élémentaires dans les interfaces qui sont : *l'ajout, la suppression et la modification* d'opérations. Les incompatibilités automatiquement résolubles sont un sous-ensemble des incompatibilités élémentaires qui possèdent des caractéristiques spécifiques qui sont discutées plus loin.

Pour la résolution des incompatibilités, l'approche par médiation de conversation est adoptée. Des médiateurs sont introduits pour contrôler les interactions entre le fournisseur et ses clients en adaptant les interfaces requises des clients avec l'interface fournie. Le mécanisme de médiation permet une résolution transparente aux clients du service. Certaines incompatibilités sont résolues en utilisant des mécanismes de temporisation et de transformation des messages reçus, avant de les faire suivre vers leur destination finale à l'aide de médiateurs [120, 29]. Les incompatibilités qui ne peuvent pas être résolues automatiquement sont retournées au client afin qu'il opère manuellement les changements nécessaires pour garantir la conformité de son interface requise avec l'interface fournie dans sa définition actuelle.

Le canevas *ArchiMed*⁴ est la principale contribution de la thèse présentée ici. Ce canevas met en œuvre la détection et la résolution des incompatibilités entre les interfaces des clients qui tentent d'interagir avec le service référencé dans le canevas [3].

Ce canevas s'appuie, entre autre, sur un module *BESERIAL* (*Behavioural Service Interface Analyser*) qui met en œuvre les algorithmes de détection des incompatibilités de base [2]. Sur le plan théorique nous avons introduit des expressions pour la formalisation des incompatibilités élémentaires. Nous avons démontré que ces expressions s'excluent mutuellement et qu'elles couvrent toutes les incompatibilités élémentaires. Ces expressions sont utilisées dans un algorithme de détection des incompatibilités entre deux interfaces P et P' . L'algorithme retourne toutes les différences entre P et P' qui font que P' ne simule pas P . L'étude théorique de l'algorithme de détection a montré que sa complexité est moins pénalisante que celle des algorithmes présentés dans des travaux similaires [100, 74]. Dans la résolution des incompatibilités nous avons identifié des cas d'incompatibilités élémentaires qui peuvent être résolus automatiquement. Ces incompatibilités sont elles aussi représentées par des expressions qui sont interprétées et évaluées. La résolution des incompatibilités s'appuie sur des médiateurs générés automatiquement. Le canevas maintient les descriptions des versions successives de l'interface du service ainsi que l'ensemble minimal de médiateurs pour que les clients puissent interagir avec le service au travers de l'une de ses versions. Une visualisation des incompatibilités entre les interfaces est rendue disponible. Pour les incompatibilités non résolues, le médiateur utilise une mesure de similarité entre les interfaces afin de découvrir un service fournisseur dont l'interface fournie est conforme avec l'interface requise du client. Sur le plan pratique, notre contribution est la définition d'une architecture logicielle du canevas *ArchiMed* qui spécifie les liens entre les différents éléments dédiés à la détection et la résolution des incompatibilités. Un prototype du canevas *ArchiMed* met en œuvre la solution proposée. La validation expérimentale du canevas, a été effectuée par le traitement d'une collection de tests qui contient les descriptions des interfaces comportementales des services. Une étude quantitative et comparative à des travaux similaires est réalisée et montre l'apport significatif de notre proposition.

1.5 Plan de la thèse

Le document est organisé comme suit. Dans la première partie de la thèse nous présentons en deux chapitres le contexte de la recherche et un état de l'art. Dans le chapitre 1, après avoir présenté le contexte sur les services web, nous définissons la problématique liée aux incompatibilités des interactions entre un service web et ses clients. Les incom-

⁴Architecture de Médiation

patibilités traitées sont liées à changements dans les définitions des interfaces fournies. Le chapitre 2 présente un état de l'art sur la problématique de compatibilité des interfaces. L'état de l'art présente les différentes approches de test de conformité des interfaces ainsi que les différentes techniques existantes pour la résolution des incompatibilités.

Dans la deuxième partie de la thèse, nous détaillons notre proposition du canevas *ArchiMed* pour la réconciliation de conversations par génération automatique de médiateurs. Le chapitre 3 introduit la modélisation formelle des interfaces en se basant sur la théorie des automates déterministes en nombre fini d'états. Les principes de la résolution des incompatibilités par médiation sont également présentés et illustrés dans ce chapitre. Le chapitre 4 détaille une des contributions principales de la thèse qui est la détection des incompatibilités élémentaires (*ajout*, *suppression* et *modification* d'opérations). Un algorithme compare les interfaces des services et retourne toutes les incompatibilités. Dans le chapitre 5, nous détaillons les cas d'incompatibilités qui peuvent être résolues automatiquement en s'appuyant sur le résultat de la détection des incompatibilités élémentaires. La résolution des incompatibilités est basée sur un mécanisme de génération automatique de l'interface du médiateur. Le chapitre 6 présente la mise en œuvre et les détails d'implantation du canevas *ArchiMed*, ainsi que les résultats des tests menés dans l'étude de compatibilité des interfaces. Enfin, le chapitre 7 conclut cette thèse sur les principales contributions mais également sur ses limites et les différentes pistes de la recherche que ce travail a permis d'identifier.

Chapitre 2

Etat de l'art

Sommaire

2.1	Introduction	25
2.2	Tests de conformité et résolution des incompatibilités . .	26
2.2.1	Comportements et structures	27
2.2.2	Compatibilité à la conception d'une composition de services	30
2.2.3	Résolution à l'exécution	39
2.3	Changements des interfaces	46
2.3.1	Patrons de changement	46
2.3.2	Évolution des instances de processus en cours d'exécution .	49
2.3.3	Versions d'une interface fournie	52
2.4	Synthèse	58
2.4.1	Structure et comportement	58
2.4.2	Détection	59
2.4.3	Résolution	60
2.4.4	Tableaux récapitulatifs	62
2.5	Conclusion	66

2.1 Introduction

Dans ce document de thèse, nous avons fixé le contexte de notre recherche et nous avons défini 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 présentons un état de l'art sur les différentes approches qui sont proposées autour de cette problématiques. Des techniques sont proposées pour

automatiser la détection et la résolution des incompatibilités entre les interfaces afin de ne pas engager des coûts que nécessite un recodage de toutes les applications clientes qui interagissent avec un service dont la définition de l'interface a changé. Des travaux ont été menés pour résoudre ce problème et une des idées proposées consiste à vérifier la conformité des interfaces en s'appuyant sur des théories formelles pour la modélisation des interfaces. Des tests de conformité et des techniques de résolution des incompatibilités sont présentés dans la section 2.2. Une deuxième démarche consiste à considérer le point de vue du client où le concepteur ne définit pas une nouvelle mise en œuvre du client qui soit conforme avec la nouvelle définition de l'interface du service. Toutefois, le concepteur cherche un autre service fournisseur dont l'interface fournie est compatible avec l'interface requise du client. Ces approches sont basées sur la découverte de services qui utilise des tests de conformité entre l'interface requise du client, posée comme requête au système de découverte de services, et les interfaces fournies, retournées comme résultats positifs aux tests de conformité (voir la section 2.2.3). D'autres approches proposent de résoudre les problèmes des incompatibilités dans un contexte intra-organisationnel où lorsque le concepteur d'un service introduit des modifications dans la définition de l'interface du service, il propage ces changements aux autres services pour éviter les incompatibilités. Cette technique de propagation de changements est explicitée dans les travaux présentés dans la section 2.3. A la fin de ce chapitre, une synthèse des approches existantes est donnée en section 2.4 et résume les apports et les limites de chaque approche par rapport à des critères qui renvoient aux objectifs de la thèse.

2.2 Tests de conformité et résolution des incompatibilités

Plusieurs approches ont été proposées pour détecter et résoudre les incompatibilités entre des services et chaque approche tente de résoudre une partie du problème selon le contexte et les critères qu'elle se fixe.

Pour la détection des incompatibilités, la plupart des approches se limitent à vérifier si une interface fournie est compatible ou non avec une interface requise. Les techniques mises en œuvre dans la détection s'arrêtent à la première incompatibilité qui fait deux interfaces ne sont pas conformes. Or, il peut exister plusieurs différences entre les interfaces comportementales qui font que deux interfaces ne sont pas conformes. Les différences entre les interfaces peuvent être dues à des changements élémentaires qui sont des ajouts, des suppressions et à des modifications d'opérations. De ce fait, plusieurs incompatibilités peuvent exister entre les interfaces fournie et requise.

Pour la résolution des incompatibilités, certaines approches prennent pour hypothèse que les incompatibilités ont été détectées manuellement et proposent de définir des adaptateurs. Ces derniers transforment les messages échangés de manière à garantir la compatibilité entre les structures des messages requis par les applications clientes et les structures des messages décrits dans la nouvelle interface fournie par le service. Dans d'autres approches, la résolution est faite à l'aide de mécanisme de substitution. Autrement dit, si le test de conformité entre une interface requise et une interface fournie est négatif, le développeur de l'application cliente va tenter de substituer l'ancien service dont la définition de l'interface a changé par un autre service. Ce dernier possède une interface fournie qui répond positivement au test de conformité avec l'interface requise de l'application cliente.

2.2.1 Aspects comportementaux et structuraux

Pour la résolution des incompatibilités structurelles, la plupart des travaux proposent des adaptateurs qui s'appuient sur des outils de transformation de données tels que XPath [56, 35], XQuery [57, 85, 36], XSLT [58, 67, 68] pour manipuler les structures des messages définis généralement dans le standard WSDL. La plupart des approches sur l'adaptation des interfaces des services web se sont basées sur les travaux fondateurs de *Yellin et Strom* sur l'adaptation de composants [120]. L'idée de base pour la mise en œuvre des adaptateurs est le maintien d'une pile qui mémorise les informations (messages) reçues à différents moments de la conversation. Une fois que tous les messages attendus, dans un format X , sont reçus, l'adaptateur opère automatiquement les transformations nécessaires pour obtenir un nouveau message, dans un format Y . L'adaptateur se charge ensuite de transmettre le nouveau message ainsi obtenu à son destinataire final. Un adaptateur permet de résoudre les incompatibilités structurelles en appliquant des règles de transformation des structures des messages envoyés et reçus [122]. Dans [9, 29], les approches proposées étudient les incompatibilités structurelles induites par les différences dans les formats de l'envoi et de la réception des messages. Les adaptateurs sont construits de manière semi-automatique en se basant sur des patrons d'incompatibilité identifiés ou en utilisant des opérateurs d'adaptation. 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 et que le client les envoie séparément, l'adaptateur dit de *fusion* est utilisé. Cet adaptateur reçoit ces informations du client par des réceptions multiples puis il crée une structure de message unique telle que définie dans l'interface du service. Une fois que la structure du message est conforme à la structure du message défini dans l'interface fournie du service, l'adaptateur envoie ce nouveau message au service. Toutefois, dans cette approche, aucun mécanisme de détection automatique des patrons d'incompa-

tibilité n'est proposé. En effet, les auteurs supposent que le concepteur des adaptateurs doit, dans un premier temps repérer manuellement le patron d'incompatibilité, en comparant les structures des messages XML, avant d'appliquer les opérateurs d'adaptation. Ce mécanisme de détection manuelle des incompatibilités a ses limites car si la définition de l'interface du service change vers une nouvelle définition de l'interface, où les structures des messages ont été modifiées, tous les adaptateurs de tous ses clients doivent eux aussi être redéfinis.

Il existe des incompatibilités structurelles entre les interfaces fournies et requises qui se traduisent également par des incompatibilités comportementales entre ces mêmes interfaces. Par exemple, soit un service qui peut recevoir le message d'une commande dont la structure contient toutes les lignes de commande et soit une application cliente qui ne peut envoyer que des messages de commande en une seule ligne. Ces interfaces fournies et requises sont incompatibles sur le plan structurel car la structure du message de commande regroupant plusieurs lignes de commandes (dans l'interface fournie) est différente de la structure du message d'une commande avec une seule ligne de commande (dans l'interface requise). De plus, la conformité des interfaces comportementales n'est pas vérifiée car l'opération de réception d'une commande (dans l'interface fournie) ne correspond pas à plusieurs opérations de réception des lignes de commandes. Dans ce cas, certaines approches proposent un adaptateur qui reçoit toutes les lignes de commandes envoyées par le client et les transforme en un seul message avant de l'envoyer vers le fournisseur. La figure 2.1 illustre un exemple de patron ou d'opérateur de résolution d'incompatibilité structurelle liée à plusieurs envois de messages *versus* une réception de message unique (MOP : *Many to One message Pattern*) [9, 29]. Le client **Magasin** envoie deux messages A et B qui sont reçus dans un premier temps par l'adaptateur. Ensuite, l'adaptateur transforme les deux messages reçus en un seul message. La structure du nouveau message contient deux sous-structures de message A et B que l'adaptateur envoie à destination du service **Entrepôt**.

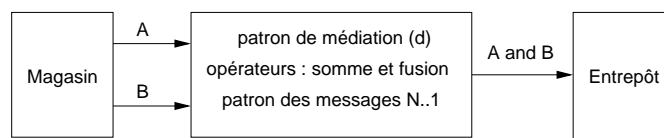


FIG. 2.1 – Adaptateur : plusieurs envois de messages *versus* réception unique de message

Dans [29], l'opérateur *fusion* est considéré comme un cas particulier de l'opérateur *somme* dans le cas où la structure de l'information à transmettre est la même. Dans la figure 2.1, l'utilisation de l'opérateur *fusion* se fait dans le cas où A et B ont la même

structure. Dans [5], les auteurs proposent la définition de médiateurs comme étant des fournisseurs virtuels. Un fournisseur virtuel transforme la structure des messages envoyés et reçus entre le service et le client. Des patrons de médiations sont présentés pour transformer les messages (voir par exemple, le patron de médiation (d)).

Dans [82], les auteurs ont repris les travaux de *Yellin* et *Strom* [120] sur l'adaptation structurelle des interfaces pour les étendre à l'adaptation comportementale de ces dernières. Les auteurs proposent des adaptateurs sur le plan structurel qui permettent de définir des mises en correspondances entre les messages échangés entre les services. Sur le plan comportemental, les auteurs proposent un algorithme qui détecte et résout simultanément les incompatibilités entre les interfaces décrites par des automates acycliques. Les modèles de processus étudiés représentent des séquences d'opérations uniquement. Le résultat de l'algorithme et la construction de l'adaptateur qui résout les interblocages entre les interfaces. La détection et la résolution est semi-automatique du fait que pour chaque incompatibilité détectée, l'algorithme sollicite le concepteur de l'adaptateur pour résoudre une incompatibilité afin de continuer la détection et la résolution. Dans la perspective de guider le concepteur dans son choix d'adaptation, un arbre dit de mise en correspondance permet de proposer des combinaisons de résolution d'une incompatibilité avec les différentes mises en correspondance des messages échangés par les services. Toutefois, l'algorithme présenté dans cette approche ne permet de détecter et de résoudre des incompatibilités entre des interfaces plus qui contiennent des choix dans des alternatives ou encore des itérations sur une ou plusieurs opérations.

Les incompatibilités comportementales résolues dans les approches ci-avant, sont déduites des incompatibilités structurelles. Un changement dans la structures des messages se traduit également par un changement dans l'enchaînement des envois et des réceptions des nouveaux messages. Cependant, sur le plan comportemental, toutes les incompatibilités ne sont pas résolues. Par exemple, si une nouvelle opération est ajoutée dans l'interface fournie d'un service et que cette opération est destinée à recevoir de nouvelles informations de clients, alors l'adaptateur est inefficace car il n'a aucun moyen de créer l'information manquante qui ne peut être envoyée que par le client. L'autre inconvénient de ces approches est que l'adaptation des interfaces est faite à la conception, or les incompatibilités peuvent surgir à la suite de la définition d'une nouvelle interface du service. Dans ce cas le concepteur du service doit définir un nouvel adaptateur à chaque nouveau changement dans la définition de l'interface du service et pour chaque client. Dans [106], les auteurs étendent ces travaux par l'introduction d'un annuaire d'adaptateurs génériques. C'est un annuaire qui regroupe des services web qui offrent les fonctions d'adaptation définis sur des protocoles standards tels que RosettaNet [98], OTA [112], HL7 [112], etc.

Lorsqu'une interface évolue, le concepteur sélectionne des adaptateurs dans l'annuaire puis il les compose selon la nouvelle description de l'interface du service. L'adaptateur composé est ensuite déployé. Cependant, un des inconvénients de cette approche est que la mise en œuvre de cette solution s'appuie sur l'utilisation de processus standards, or les processus définis dans le web ne sont pas tous standards. De plus, aucun mécanisme de détection des incompatibilités n'a été proposé dans ces approches.

2.2.2 Compatibilité à la conception d'une composition de services

Le test de la compatibilité entre les différents acteurs d'une composition de services peut être effectué au moment de la conception et/ou au moment de l'exécution de la composition de services. Dans les approches présentées dans cette section, c'est au moment de la conception de la composition de services que sont définis les partenaires qui sont compatibles et qui vont participer à l'exécution du service composite. Ces approches ne traitent pas les incompatibilités entre les services lors de l'exécution de la composition.

Dans la composition de services, les travaux proposés pour le test de compatibilité sur le plan de la description comportementale des services se basent essentiellement sur des standards tels que WSCI (*Web Service Choreography Interface*) [4, 90] et BPEL (*Business Process Execution Language*) [116]. Dans ces approches, le service composite est défini par des liens de partenariat avec les services composants pour garantir la cohérence dans les conversations entre les différents partenaires. Les interfaces fournies et requises des partenaires d'une composition de services définissent un contrat qui lie les partenaires. D'autres approches proposent de traduire les patrons de diagrammes de tâches [108] et les patrons de communication [14] en processus web exécutables dans un standard tel que BPEL (voir par exemple [116]). La conversation entre les différents partenaires impliqués dans le processus métier est orchestrée par le service composite qui possède une vision globale de l'exécution du processus. Une approche basée sur les modèles théoriques de la gestion du dialogue, propose l'utilisation des arbres de plans pour la modélisation des interfaces comportementales [7]. Dans ces arbres sont indiqués tous les chemins d'une conversation qui structure les enchaînements des actions pour aboutir à un dialogue cohérent. Toutefois, les standards tels que BPEL et WSCI sont les plus adoptés dans la description des interfaces comportementales. Une étude des standards démontre ce fait dans [116].

Si un service composant disparaît définitivement d'une composition de services, il faut le substituer par un autre service équivalent en s'appuyant sur d'autres techniques développées dans d'autres travaux (par exemple, les communautés de services [102], la découverte de services [118, 99], etc.). Dans [83], les auteurs proposent l'utilisation d'une mesure de similarités entre les graphes (qui modélisent les comportements des interfaces)

avec prise en compte de la sémantique de simulation (cette même fonction de mesure de similarité a été proposée dans [100]). La solution proposée vise à tester si un partenaire est compatible avec la composition de service. Si le résultat n'est pas satisfaisant, le développeur substitue le service par un autre, réitère le test de similarité jusqu'à aboutir à une composition cohérente. De ce fait, ce processus est semi-automatique. Dans la fonction de mesure de similarité, les cas d'évolution considérés sont uniquement la suppression d'opération et l'ajout d'opération. La modification d'opération n'est pas prise en compte. De plus, l'hypothèse que les processus considérés pour le test de la mesure de similarité n'admettent pas de cycles, ne s'applique pas à des cas concrets.

En partant des descriptions des interfaces comportementales des services en BPEL ou en WSCI, certaines approches proposent de transformer ces descriptions en représentations formelles, telles que des réseaux de Petri [19, 42, 79], des automates [15, 34, 61, 89] ou des automates temporisés [38, 37, 91]. Dans [12], les auteurs proposent d'utiliser l'algèbre des processus pour décrire les interfaces comportementales des services et pour tester la compatibilité des interfaces. Dans [32], les auteurs proposent de transformer des diagrammes de séquences de la notation UML (séquences d'envois et de réceptions de messages) en automates qui décrivent les comportements des interfaces afin d'effectuer les tests de conformité entre les interfaces. Le but de ces transformations est double. D'une part, cette représentation formelle, qui s'appuie sur la théorie des graphes, rend possible les tests de conformité et de calcul d'équivalences comportementales [13, 117]. Le calcul d'équivalence des interfaces permet de vérifier si un service peut substituer un autre service équivalent tout en gardant la cohérence de la conversation dans la composition. D'autre part, la représentation graphique du comportement des interfaces des services permet d'offrir un support visuel plus facile à manier et à comprendre par un utilisateur humain (en l'occurrence le concepteur). Ce support visuel sert à guider le concepteur dans l'objectif d'une résolution semi-automatique des incompatibilités.

Modélisation des interfaces avec des réseaux de Petri

Par définition, un réseau de Petri $N = (P, T, F)$ est un ensemble de places P , un ensemble de transitions T et une relation de flux F (arcs) [77]. La transition représente l'élément dynamique, cela peut modéliser une activité d'un processus métier (par exemple, recevoir une commande d'un client). La place représente l'élément statique qui définit les dépendances entre les activités. L'état d'un réseau de Petri est représenté par l'ensemble des jetons répartis sur les places. Autrement dit, c'est le marquage du réseau qui désigne son état à un instant donné. Chaque arc relie exactement une place à une transition. La

transition ne peut être déclenchée que si toutes les places qui la précèdent sont marquées par au moins un jeton. Après le déclenchement de la transition, un jeton est consommé de chacune de ses places précédentes, puis chacune des places qui suivent la transition, sera marquée d'un jeton supplémentaire.

Dans les approches à base de réseaux de Petri [27, 19, 76, 77], les algorithmes de bisimilarité permettent de détecter les incompatibilités comportementales avec l'interface attendue par le service client. En cas d'incompatibilité, une trace de la première erreur est retournée au concepteur sur les graphes donnés en entrées. Etant donné qu'une seule incompatibilité est détectée à la fois, une autre itération pour la détection de l'incompatibilité suivante (si elle existe) est nécessaire, après la résolution de l'incompatibilité précédente. Mais aussi, ces approches se limitent à la résolution des incompatibilités à la conception et ne prennent pas en compte l'évolution des interfaces.

Dans [42], la transformation de BPEL en un réseau de Petri est démontrée possible. Cependant, d'après les auteurs la complexité des algorithmes (compatibilité, bisimulation, atteignabilité) appliqués aux réseaux de Petri obtenue est plus importante à cause des explosions combinatoires. Les réseaux de Petri dédiés à la modélisation des processus métiers se distinguent par une place initiale qui dénote le début du processus, et une place finale qui indique la fin du processus. Afin de traduire des processus BPEL en réseaux de Petri, des patrons génériques sont proposés pour modéliser les activités élémentaires (par exemple, l'activité *invoke* qui permet de solliciter une opération offerte par un service) et les activités structurelles (par exemple, l'activité *sequence* qui permet de définir une séquence d'activités élémentaires ou structurelles). Une fois les descriptions des interfaces comportementales en BPEL transformées en réseaux de Petri, des algorithmes de vérification de la compatibilité des interfaces peuvent être utilisés. Ces tests de compatibilité concernent l'aspect structurel où l'interface requise d'un client doit décrire des opérations d'envoi et de réception de messages telles que décrites dans l'interface fournie. Au niveau des tests de conformité des interfaces sur le plan comportemental, la simulation de la composition des deux interfaces de deux partenaires ne doit pas induire de situation d'inter-blocage. L'inter-blocage désigne la situation où chaque partenaire est en attente mutuelle de réception ou d'envoi d'un message de l'autre partenaire pour continuer l'exécution de la conversation [8]. Dans ce cas l'attente est infinie et engendre l'inter-blocage sans que la conversation ne puisse se terminer. Des outils pour la détection des inter-blocages et le calcul des graphes d'atteignabilité tels que WofBpel [86] et Lola [77] sont utilisés. Toutefois, les outils permettent uniquement de détecter s'il existe une incompatibilité sans pour autant la localiser sur les deux graphes. Si d'autres incompatibilités existent, les algorithmes proposés ne peuvent pas les détecter car ils s'arrêtent dès la première incompatibilité retrouvée.

Une autre approche propose de définir différents points de vue dans la conception et la réalisation de services web composites [27]. Dans cette dernière, la modélisation de composition considérée est la choregraphie. Les interfaces comportementales sont définies à l'aide de réseaux de Petri. Une fois les différents partenaires de la composition identifiés, les interactions respectives entre eux sont définies. Des tests de conformité entre les interfaces des services entrant dans la composition permettent d'identifier si une interface fournie n'est pas compatible avec une interface requise. Les auteurs définissent un opérateur de composition des réseaux de Petri qui modélisent les interfaces des services. Pour vérifier la compatibilité des interfaces, les auteurs s'appuient sur des comparaisons de traces générées à partir des réseaux de Petri qui modélisent les interfaces des services. Or, les traces peuvent ne pas être toutes générées dans la cas de réseaux de Petri à état infinis.

Dans les approches citées précédemment, la modélisation du comportement d'un service web par des réseaux de Petri permet de modéliser les opérations et les enchaînements entre ces différentes opérations. La détection des incompatibilités est restreinte à la détection de la première incompatibilité retrouvée par l'algorithme de test de bisimilarité. Une fois une incompatibilité est détectée, le concepteur doit la localiser sur les deux graphes et identifier manuellement le patron d'incompatibilité dont il s'agit. Les incompatibilités résolues sont restreintes aux cas des incompatibilités structurelles. Les patrons de résolution proposés comme adaptateurs permettent la transformation de messages envoyés et reçus afin de garantir la compatibilité structurelle. Cependant, les adaptateurs ne prennent pas en compte les nouvelles incompatibilités qui apparaissent suite au changement de la définition de l'interface du service.

Modélisation des interfaces avec des automates :

Un autre outil théorique pour la modélisation du comportement des interfaces des services web est la théorie des automates déterministes en nombre fini d'états [89]. Un automate déterministe en nombre fini d'états est un quintuplet $(E, \Sigma, \rightarrow, e_0, F)$ où :

- E : l'ensemble des états
- Σ : l'ensemble des symboles des messages en envoi et en réception,
- \rightarrow : la fonction de transition qui prend en arguments un état (état source) et un message d'entrée et retourne un état (état cible).
- e_0 : l'état initial avec $e_0 \in E$
- F : l'ensemble des états finals avec $F \subset E$

Les transitions entre les états sont définies par des flèches étiquetées par des messages. Les messages ont une polarité, ceux envoyés sont précédés du symbole $>$ tandis que ceux reçus sont précédés du symbole $<$. Chaque instance du processus est associée à une

instance de l'automate.

Un exemple est illustré dans la figure 2.2 où un processus de gestion de commande est modélisé sous forme d'un automate. Les rectangles représentent les états de l'automate (voir par exemple les états *Commande*, *Paiement*, etc.). Les flèches représentent les transitions qui partent d'un état source vers un état cible et qui sont étiquetées d'un événement d'envoi ou de réception d'un message. Par exemple la flèche qui relie l'état source *Commande* à l'état cible *Paiement* est étiquetée d'un envoi de message de confirmation. Une garde peut être associée à une transition (voir par exemple, la garde qui spécifie que la quantité commandée doit être disponible pour déclencher la transition de l'état *Commande* à l'état *Paiement*). L'état initial est représenté par un disque noir et les états finals sont représentés par un disque dans un cercle.

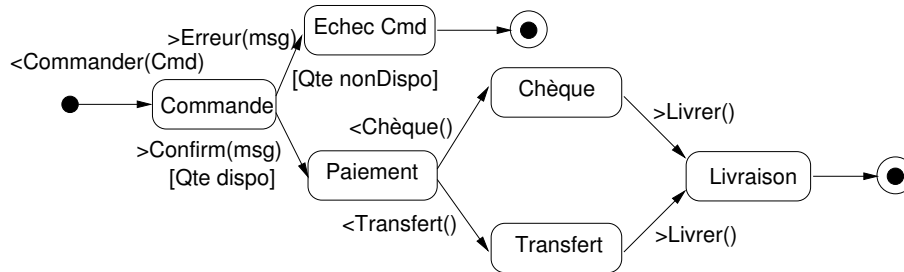


FIG. 2.2 – Représentation en automate d'un processus de commande

Dans l'exemple de la figure 2.2, après la réception du message de commande d'un client, l'automate passe de l'état initial vers l'état *Commande*. Si la quantité des articles commandés est disponible, alors un message de confirmation est envoyé au client et l'automate passe dans l'état *Paiement*. Dans le cas contraire, c'est un message d'erreur qui est envoyé au client et la conversation est terminée. Dans l'état *Paiement*, l'automate attend la réception d'un message du client pour effectuer le paiement de la commande soit par l'envoi d'un chèque, soit par l'envoi d'un ordre de transfert bancaire. A la réception d'un message de transfert, l'automate passe dans l'état *Transfert*. Puis, un message de livraison est envoyé au client et l'automate passe dans l'état final *Livraison* ce qui termine la conversation.

Les apports des approches à base d'automates résident dans la proposition d'algorithmes de tests de conformité entre les interfaces fournies et les interfaces requises. Il s'agit en effet d'utiliser les représentations en automates des interfaces fournies et requises afin d'appliquer ensuite les algorithmes de test de conformité qui existent sur les automates. Des tests de simulation entre deux interfaces fournies sont également possibles. Dans [89], les auteurs proposent de s'appuyer sur des algorithmes de tests de conformité pour aider les concepteurs dans la composition de nouveaux services web. Lorsqu'une

incompatibilité est détectée, elle est signalée au concepteur qui doit modifier les liens des interactions entre le client et le service pour résoudre l'incompatibilité. Ensuite, un autre test est effectué sur la nouvelle composition de services proposée par le concepteur et ainsi de suite jusqu'à obtenir une composition de services dont le schéma de conversation est cohérent. En d'autres termes, c'est un mécanisme itératif de détection et de résolution d'incompatibilité. Le concepteur réitère la détection et la résolution d'une incompatibilité, l'une après l'autre, jusqu'à ce que la conformité entre les interfaces fournies et requises des partenaires soit vérifiée.

Dans [15, 34, 61], les auteurs étendent la modélisation des interfaces comportementales des services par l'introduction d'une pile qui mémorise les messages en entrée et en sortie. L'automate ainsi obtenu est dit *machine de Mealy* [15]. Les algorithmes proposés pour les tests de conformité des interfaces maintiennent une pile des messages reçus. A chaque fois qu'une interface sollicite la réception d'un message, l'algorithme vérifie dans la pile si le message n'a pas été déjà reçu. Si c'est le cas, ce message est dépilé pour être consommé par l'interface qui le sollicite. L'incompatibilité entre deux interfaces est détectée lorsque le message requis dans une interface n'apparaît pas dans l'interface sollicitée et que la pile ne contient pas ce message. Dans ce cas, l'algorithme s'arrête et retourne la première incompatibilité détectée mais ne retourne pas toutes les incompatibilités qui peuvent exister.

Dans la composition de services, les auteurs proposent des opérateurs de composition qui prennent en compte les aspects structurels des messages échangés et les aspects comportementaux liés à l'enchaînement des envois et des réceptions des messages [83]. Une extension des algorithmes de bissimulation est introduite en affectant un poids au couple d'états considéré en fonction des transitions en communs, de manière quasi identique aux proposition [100, 23]. La différence réside dans le fait qu'ils considèrent en plus les aspects intégration de données, en s'appuyant sur des techniques d'analyse textuelle avec des algorithmes dits N-gramme. Le résultat retourné par les algorithmes sont des valeurs numériques qui indiquent le degré de similarité entre deux interfaces. Cette approche ne fournit pas les positions des incompatibilités dans les graphes comparés dans la perspective de les résoudre.

Les approches à base d'automates proposent des algorithmes de test de conformité entre les interfaces fournies et les interfaces requises afin de les intégrer dans des outils de conception pour la composition de services (selon le modèle d'orchestration ou de chorégraphie). La conception est alors guidée par la détection des incompatibilités comportementales qui sont signalées au concepteur qui doit les résoudre de lui-même (semi-automatique). Les algorithmes proposés permettent uniquement de répondre si la compatibilité est garantie ou non. La localisation des incompatibilités sur les graphes n'est

pas faite. Le concepteur procède par essai/erreur en visualisant globalement les interfaces. Une fois la compatibilité entre les différents partenaires de la composition garantie, le nouveau service est déployé. Cependant, la conformité entre les interfaces fournies et requises n'est pas toujours vérifiée lorsque les interfaces des services entrants dans la composition changent. De ce fait, chaque service doit prendre en compte les changements des interfaces des autres services. En d'autres termes, chaque interface comportementale d'un service doit être adaptée à chaque nouveau changement de l'interface d'un autre service qui induit des incohérences comportementales dans les conversations. Il en va de même pour les concepteurs des applications clientes qui doivent prendre en compte le nouveau comportement du service composite.

Outils de modélisation :

Les outils existants, pour la modélisation de processus et pour la composition de services, offrent des fonctionnalités qui permettent de concevoir et de déployer un nouveau processus ou une nouvelle composition de services. *BizTalk* est un composant de la plateforme applicative de *Microsoft* pour automatiser et optimiser les processus métiers [51, 41]. *BizTalk* permet de mettre en œuvre un système d'information à base de services, en intégrant les applications des partenaires et des collaborateurs qui interviennent dans les processus métiers. Des outils graphiques permettent de définir les modèles des processus métiers. Une fois le modèle d'un processus est validé (pas d'incompatibilités entre les partenaires), il est déployé dans un moteur d'exécution.

Le langage de modélisation *YAWL* (*Yet Another Workflow Language*), permet de définir et des processus métiers qui seront déployés dans un moteur d'exécution de processus [60, 107, 104]. L'outil proposé transforme les modèles de processus décrits en *YAWL* en représentation formelle basée sur la théorie des réseaux de Petri. Cette modélisation des processus en réseaux de Petri est utilisée dans des tests de cohérence dans les définitions des enchaînements des activités.

WebSphere est un outil proposé par *IBM* qui permet de construire des processus métiers à base de description *BPEL* et *WSDL* des partenaires qui entrent dans l'orchestration [48, 40, 121]. Un moteur d'exécution de processus permet d'instancier le modèle du processus décrit en *BPEL* et de le rendre exécutable.

Cependant, tous ces outils n'offrent pas la possibilité de garantir la cohérence des interactions avec les clients si le modèle du processus évolue. Lorsqu'un développeur modifie un modèle processus métier pour en obtenir une nouvelle description du modèle, l'outil ne prend pas en compte le fait que ces évolutions peuvent induire des incompatibilités avec les clients et les partenaires qui sollicitent les opérations que le processus proposait dans

l'ancienne définition du modèle de processus. De plus, ces outils ne vérifient la cohérence des interactions et des conversations entre les partenaires qu'à la conception. Une fois les instances de processus déployées, il n'existe aucun moyen de propager la description des évolutions du modèle aux instances de processus en cours d'exécution.

Sélection de services compatibles à la conception :

L'objectif des travaux présentés ici est de sélectionner, au moment de la conception d'une composition, les services qui doivent participer à cette composition. Des tests de similarité entre les interfaces comportementales des services sont proposés afin de choisir les services qui participent à la composition de services.

Les tests de similarité sur des graphes sont très utilisés dans divers domaines tels que la reconnaissance de patrons de graphe [96]. Des mesures de distances entre les graphes sont introduites et s'inspirent des travaux sur les mesures des distances entre les chaînes de caractères [113]. L'objectif est de retrouver le nombre minimum de modifications à effectuer (ajout, suppression et modification des arêtes et des nœuds) pour garantir un isomorphisme de graphe [18]. Dans [73], les auteurs proposent de transformer les descriptions des interfaces comportementales en structures d'arbres. Les structures d'arbres sont comparées pour en extraire les différences.

Cependant, dans cette démarche, l'isomorphisme de graphe n'est garantie que sur un plan strictement structurel et ne prend pas en compte la sémantique comportementale des graphes. En d'autres termes, la simulation du comportement entre graphes n'est pas garantie.

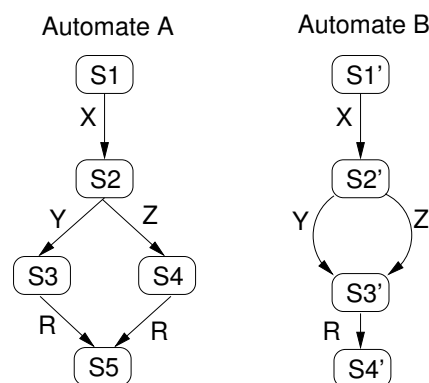


FIG. 2.3 – Similarité : structure de graphe et sémantique de comportement

L'exemple de la figure 2.3 montre deux automates A et B dont les structures en graphes sont différentes. L'automate A contient un état supplémentaire (S4) et une transition

supplémentaire entre S4 et S5 étiquetée par R. Cependant, le comportement de A et de B sont identiques car tous les deux simulent les deux séquences suivantes : 1) X puis Y puis R et 2) X puis Z puis R.

Pour détecter les différences entre les comportements modélisés par deux automates une idée est de comparer les expressions acceptées par chacun des deux automates. Dans [22], l'auteur présente la théorie de *Moore* qui est basée sur des expérimentations ou tests pour vérifier si un automate A a été modifié en un nouvel automate B. *Moore* construit une fonction de test qui prend en paramètre toutes les séquences de mots (acceptées par l'automate A) qui sont prises en entrée du nouvel automate B, et détecte si la séquence a été étendue par l'ajout d'un mot ou bien si des mots ont été supprimés. Dans [113], les auteurs s'appuient sur des tests similaires pour détecter les ajouts, les suppressions et les modifications de mots entre deux chaînes de caractères. Cependant, les séquences considérées sont toutes de tailles prédéfinies car les expressions régulières ne sont pas prises en compte. Cette modélisation ne prend pas en compte la définition des cycles dans les automates. La théorie de *Kleene* sur les événements et les expressions régulières annonce dans un premier temps que tout automate admet une expression régulière qui reconnaît les mêmes séquences de mots admises par l'automate [22]. Cependant, plusieurs expressions régulières peuvent être associées à un même automate. Pour tester l'équivalence de deux automates A et B, l'auteur propose de vérifier si un isomorphisme de graphes existe entre A et B. Cependant, ce test d'équivalence ne prend pas en compte que deux automates soient équivalents sur le plan comportementale (bisimulation) sans que l'isomorphisme de graphes entre les automates A et B soit vérifié.

D'autres approches proposent des mesures de similarité qui prennent en compte le comportement des interfaces des services modélisées avec des automates. La mesure de similarité est définie pour un couple d'états initiaux du système composé par les deux automates à comparer. C'est une mesure calculée récursivement et qui attribue une valeur à chaque couple d'états, chaque état pris dans l'un des deux automates à comparer. La valeur est calculée en fonction des transitions sortantes de chaque état du couple considéré. Les transitions qui engendrent des différences sont considérées comme des ε -transitions (ou *stuttering transitions*) qui pénalisent la valeur de similarité dans un couple d'états courant [100].

L'exemple de la figure 2.4 illustre un couple d'états (S1,S2) tel que S1 est un état d'un automate A1 et S2 est un état d'un autre automate A2. L'état S1 possède une transition avec une étiquette R qui n'apparaît pas les transitions sortantes de S2. Dans ce cas, la valeur de la distance entre A1 et A2 augmente du fait que R a été supprimée de S2. La transition d'étiquette R est considérée comme une ε -transition, donc le calcul de la distance entre A1 et A2 va continuer en considérant le couple d'états S1 et l'état cible de

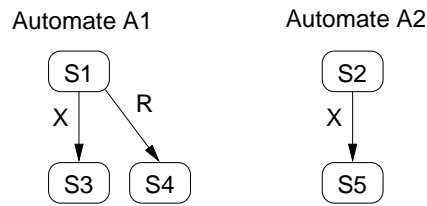


FIG. 2.4 – Mesure de similarité entre deux automates

la ε -transition qui est S5. Un autre couple d'états à considérer dans le calcul de la mesure de similarité entre A1 et A2 est le couple (S3,S5). Ce dernier est obtenu en calculant les états cibles des transitions sortantes des états S1 et S2 qui ont le libellé X en commun.

Dans [23, 74], les auteurs proposent également d'étendre la mesure de similarité entre graphes par une autre mesure qui prend en compte les aspects comportementaux. Cette mesure de similarité est ensuite utilisée dans la découverte de services web dont la description en BPEL est conforme à la requête ou à l'interface requise par le client qui cherche ce service. Cependant, dans les fonctions de mesure de distance de similarité, les interfaces comportementales modélisées ne contiennent pas de cycles. Dans ces approches, le cas de la mesure de similarités entre deux automates qui contiennent des cycles ne se terminent pas. De plus, il n'y a aucun moyen de retourner avec précision les états, dans les deux automates comparés, où les incompatibilités apparaissent (dans la perspective de résoudre ces incompatibilités).

2.2.3 Résolution des incompatibilités à l'exécution

Dans la section précédente, nous avons discuté de la vérification de la compatibilité à la conception d'une composition de services entre les partenaires eux-mêmes et entre les partenaires et les clients. Cependant, lorsque la définition de l'interface d'un service change la compatibilité entre les partenaires peut être rompue. Ainsi, la conformité qui est vérifiée à la conception du schéma de conversation dans une composition de services peut ne pas être vérifiée à l'exécution de celle-ci.

Les annuaires de services :

Un service client peut ne pas connaître à l'avance le service fournisseur avec lequel il va interagir au moment de l'exécution. Ce type d'interaction nécessite la découverte de services fournisseurs qui répondent aux besoins des clients en termes d'opérations et de fonctionnalités recherchées. Comme illustré dans la figure 2.5, les services fournisseurs s'enregistrent dans un annuaire et renseignent leurs descriptions dans le format WSDL

ainsi que d'autres informations telle qu'une description textuelle des fonctionnalités du service. Le client envoie une requête à l'annuaire. Cette requête désigne l'interface requise du client. L'annuaire, par des mécanismes de test de conformité entre l'interface requise posée en requête par le client, et les interfaces fournies des services abonnés à l'annuaire, retourne les informations sur les services fournisseurs qui sont conformes à la requête. Une sélection peut être appliquée à l'ensemble des services fournisseurs retournés selon des critères de qualité prédéfinis dans la requête du client. Le client va ensuite interagir avec le service fournisseur choisi en respectant la description des opérations données dans la description en format WSDL de l'interface fournie par le service [80].

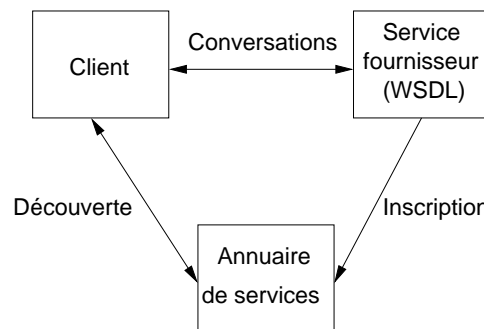


FIG. 2.5 – Architecture de l'annuaire de services web

Dans [10, 102, 75], les auteurs proposent de regrouper des services qui offrent les mêmes opérations en communautés. La communauté de services peut être considéré comme un annuaire de services spécialisé qui regroupe des services qui offrent les mêmes opérations [102, 28]. Les services qui appartiennent à une communauté sont équivalents sur le plan structurel. En d'autres termes, les structures des messages envoyés et reçus par tous les services sont les mêmes. Si l'équivalence structurelle n'est pas vérifiée, des adaptateurs sont introduits pour assurer la transformation des messages dans la structure appropriée. Le principe d'utilisation d'une communauté est : lors de l'exécution d'une composition de services ou encore lorsqu'un client sollicite une opération de la communauté, un des services abonnés à cette communauté est choisi pour répondre à la requête. La requête porte par exemple des critères de qualité des services tels que la disponibilité du service et le temps de réponse [16]. Dans [102], les auteurs définissent un objet OSC (*Open Service Connectivity*) du service sélectionné depuis la communauté de services. L'objet OSC est fourni au client afin qu'il puisse garantir la mise en correspondance structurelle des opérations à l'exécution.

Cependant, ces dernières approches, ne prennent pas en compte la définition des interfaces comportementales des services abonnés à une communauté de services. En effet, rien n'est mentionné quant à la possibilité de substituer un service par un autre service sur le

plan comportemental. Si la cohérence comportementale est garantie entre un service client et un service abonné à la communauté, elle risque de ne pas toujours être garantie lors d'une prochaine exécution avec un autre service abonné. L'équivalence entre les interfaces comportementales des services abonnés à la communauté n'est pas vérifiée.

Prise en compte de la sémantique :

Certaines approches proposent d'ajouter une description sémantique à base de logique descriptive (OWL-S, RDF, RDFS) sur les descriptions WSDL et proposent des techniques d'interrogation et de recherche connues du domaine de la Recherche d'Information. Dans [101], les auteurs décrivent les services dans le format WSMO (*Web Service Modeling Ontology*) qui est une description dans le format WSDL du service à laquelle est reliée un ontologie des concepts manipulés dans le service. Les auteurs introduisent également des Delta-relations (Δ -relations) qui définissent des relations de généralisation/spécialisation entre les concepts d'une ontologie pour calculer des différences sémantiques entre les services web et les buts (requêtes) qu'ils peuvent satisfaire. Ces relations sont appliquées aux descriptions WSMO afin de permettre des raisonnements avec des règles d'inférence. Ainsi, dans la formulation des requêtes des services clients il est possible de raffiner la découverte des services web grâce aux relations qui peuvent relier les différentes capacités de services. Des règles d'inférence peuvent être appliquées sur des résultats globaux (par exemple, trouver les services web qui renseignent sur tous les restaurants de France) pour répondre à des requêtes plus fines (par exemple, trouver les services web qui renseignent sur les restaurants de catégorie deux étoiles en France). Toutefois, cette approche ne traite que la découverte de services selon leurs aspects structuraux. Les aspects comportementaux relatifs à la description de processus métiers (par exemple, décrits en BPEL) ne sont pas traités par les auteurs.

Dans la communauté du web sémantique, la découverte de services web compatibles avec les interfaces requises par les services clients est rendue possible grâce à des annotations sémantiques ajoutées à la description des services. Ces annotations décrivent les caractéristiques des services et les actions qu'ils réalisent à l'aide d'ontologies d'un domaine donné. Dans ce contexte, une ontologie est un ensemble d'informations dans lequel sont définis les concepts utilisés dans un domaine particulier et qui décrit les relations logiques entre eux. Un standard tel que DAML-S [49] est utilisé pour ajouter une description sémantique aux services web [6]. Par la suite, DAML-S a évolué en OWL-S [50].

La description du service contient le profil du service qui décrit les opérations, leurs paramètres et leurs type qui sont des caractéristiques structurelles, ainsi que le modèle qui spécifie les aspects dynamiques du service (le processus). La découverte de services par un

client se fait uniquement sur le profil par un mécanisme de généralisation/spécialisation. Une fois le service retourné, le client initie le processus du service selon la description offerte par le modèle. Cependant, le client peut avoir prédéfini un comportement qu'il requiert mais qui n'est pas compatible avec le comportement fourni par le modèle du service résultat de la requête. Par exemple, dans un contexte de vente d'articles, le service client attend d'être livré avant d'effectuer le paiement tandis que le fournisseur attend le paiement avant de livrer son client. Cette conversation engendre un interblocage d'où l'intérêt de vérifier également la compatibilité des interfaces sur le plan comportemental.

Dans [66], les auteurs proposent d'étendre le standard UDDI par des descriptions sémantiques en DAML-S afin d'introduire une phase de découverte sémantique de services. Des indicateurs issues de la recherche d'information (par exemple, la fréquence des termes dans une description) sont également utilisées pour cerner les aspects non sémantiques. Le mécanisme de découverte dans un annuaire UDDI est étendu par une recherche par mots clés extraits de *TModels*. Les *TModels* sont des représentations sous forme d'ontologies des propriétés structurelles et comportementales des services [49]. Une approche similaire est présentée dans [88] où les descriptions des services enregistrés dans un annuaire UDDI sont traduites en descriptions de leur profil en DAML-S. En partant de ces profils basés sur une ontologie donnée, la requête du service client est traduite par un mécanisme de spécialisation / généralisation (*subsumption*), appliqué aux structures des messages, qui retourne soit un résultat exact soit un résultat générique ou spécifique.

Les approches ci-avant qui se basent sur des descriptions sémantiques des services web pour la découverte de services prennent pour hypothèse l'existence d'une ontologie commune. Cette dernière définit des frontières précises entre les différents domaines et contextes existant dans le web. Une fois le domaine fixé, la découverte de services qui se rapprochent le plus possible sémantiquement de la requête est réalisée par un mécanisme d'inférence. Cependant, l'hypothèse forte d'une ontologie commune à tout le web est peu réaliste. De plus, les requêtes exprimées ne considèrent que les aspects liés à la structure des services web. La conformité sur le plan comportemental n'est pas discutée.

Dans [99], les auteurs proposent de prendre en compte les aspects comportementaux dans la découverte de services. Il s'agit d'une modélisation en automates des comportements et des processus offerts par un service fournisseur. Les requêtes des clients, qui sont définies par leurs interfaces requises, sont également décrites en automates. Chaque requête modélise le comportement de l'interface attendue par le client. Les auteurs ont proposé un langage d'interrogation à base de six patrons de comportements qui permettent de décrire des enchaînements d'opérations. Par exemple : *Avant(A,B)* est une requête qui retourne toutes les opérations *A* qui s'exécutent avant *B*. Ces patrons peuvent être com-

binés dans des expressions. A l'aide de ce langage d'interrogation, le service client formule ses requêtes qui décrivent le comportement attendu. La requête initiale est transformée en automate déterministe en nombre d'états fini. L'objectif est de comparer les expressions qui sont générées à partir de l'automate de la requête avec les expressions générées à partir des automates des services fournisseurs. Si les expressions générées par l'automate de la requête sont incluses dans les expressions générées par l'automate d'un fournisseur alors l'interface comportementale fournie est compatible avec l'interface requise par le client.

La découverte de services est un mécanisme qui est utilisé par les concepteurs des clients lorsque les interfaces requises des applications clientes ne sont plus compatibles avec l'interface fournie du service fournisseur. En effet, lorsque la définition de l'interface fournie change elle engendre des incompatibilités avec les interfaces requises des clients. Le concepteur du client sélectionne un autre service, parmi les résultats de la découverte de services dont les interfaces fournies sont compatibles avec l'interface requise. Dans [1], les auteurs proposent une approche basée sur la logique temporelle pour la découverte de services en introduisant un langage de requêtes pour les services web (*WSRL : Web Service Request Language*). La modélisation de composition de services selon des orchestrations est offerte. Les services qui entrent dans une orchestration sont sélectionnés. Les critères de sélection portent sur les propriétés non fonctionnelles des services. Ces dernières sont définies par le client qui va interagir avec le service composite. Le langage WSRL est utilisé pour la formalisation des requêtes qui retournent l'ensembles des services qui entrent dans la composition. Un mécanisme de planification des services web est ensuite utilisé qui consiste à définir un plan décrivant les enchaînements d'interactions dans la perspective d'atteindre le but attendu par le client. L'utilisateur qui formule la requête initiale a la possibilité de la reformuler si les plans proposés en résultat ne satisfont complètement ses besoins.

Dans cette proposition, les services, enregistrés dans un annuaire UDDI, sont supposés respecter un standard qui fixe le format des interactions dans un domaine donné (par exemple, OTA [52], RosettaNet [46], etc.). Ceci garantit la compatibilité comportementale entre la requête (le but) de l'utilisateur et le résultat (le plan) instancié via UDDI. Toutefois, le recours à des processus standardisés dans la définition des interfaces des services web restreint le champ d'application de cette proposition. Les évolutions et les changements des interfaces traités dans cette approche sont relatifs aux propriétés non fonctionnelles des services. La résolution consiste en un mécanisme de formulation de nouvelles requêtes par un utilisateur jusqu'à trouver la composition de services qui satisfait les nouveaux besoins non fonctionnels des clients.

Sélection de services compatibles à l'exécution

De manière générale, dans les études sur la conformité des interfaces comportementales, l'accent est mis sur la cohérence des enchaînements des envois et des réceptions de messages entre le service et le client. L'interface du service décrit des enchaînements d'interactions (par exemple, en BPEL). Les applications clientes sont mise en œuvre de manière à ce que leurs interfaces requises soient conformes avec l'interface fournie par le service. La découverte de services dont les interfaces fournies sont conformes sur le plan comportemental à l'interface requise d'un client est une problématique abordée dans plusieurs travaux [13, 117, 89]. Le principe de base consiste à transformer la description textuelle du comportement fournie en BPEL ou WSCI par exemple, en une description qui s'appuient sur des notations formelles (automates, réseaux de Petri, etc.), puis d'utiliser les méthodes de vérification de conformité offertes par ces théories. Les services conformes à l'interface requise par le client sont retournés en résultat et parmi les services retournés, d'autres critères de choix peuvent être appliqués (par exemple, des critères de qualité de service tels que le prix, la disponibilité, le temps de réponse, etc.) afin de n'en retenir qu'un lors de l'exécution. Dans cette démarche, le client ne connaît les services avec lesquels il va interagir qu'au moment de l'exécution. Ainsi, les services dont les définitions des interfaces fournies ont changé et qui ne sont plus conformes avec l'interface requise par le client, ne seront pas sélectionnés.

Dans [78], une composition de services, selon une orchestration, est vue comme un ensemble de spécifications structurelles (les opérations) et comportementales (l'enchaînement des opérations) associées à chaque partenaire. Les interfaces fournies des services qui interviennent dans une composition sont vues comme des contrats. En cas de changement de la définition d'une interface fournie, le concepteur du service doit maintenir la compatibilité de la composition. Deux solutions sont possibles : i) maintenir les spécifications de l'ancienne définition de l'interface, ii) mettre en œuvre une nouvelle composition de services avec la prise en compte de la nouvelle définition de l'interface fournie du service qui a changé. La définition d'une nouvelle composition de services implique la mise en œuvre de nouveaux liens d'interaction entre les services entrants dans la composition et les clients qui l'utilisent. La description conceptuelle d'un service est donnée sous forme de diagrammes UML (diagramme de classes pour la description des aspects structurels, et diagrammes d'états pour la description des aspects comportementaux). Les descriptions des services sont stockées dans un annuaire qui contient également les descriptions des processus de coopération entre les différents services. Si un nouveau partenaire décide de s'enregistrer dans cet annuaire, il doit adapter son interface au processus auquel il désire s'inscrire. La substitution d'un service dans une composition est possible dans le cas où

le nouveau service inclut le comportement et la structure du service substitué ceci afin de garantir la compatibilité du processus. La substitution n'est pas possible lorsqu'une instance du service à substituer est en cours d'exécution. Le calcul de la compatibilité entre le nouveau service substitut et le processus de coopération est effectué, *a posteriori*, en vérifiant que chaque trace générée par le service substitué peut être générée par le service qui le substitue. Une trace est une séquence de messages d'envoi et de réception de messages que produit un service dans le processus global.

Une des hypothèses de cette approche est que les services fournisseurs sont abonnés à un processus métier dont la description ne change pas. En cas d'évolution de l'interface d'un partenaire vers une nouvelle définition de son interface fournie, la compatibilité de ce dernier avec le processus global doit être vérifiée à nouveau. Si la nouvelle interface obtenue n'est plus compatible avec l'interface globale, le service sera substitué par un autre dont l'interface fournie satisfait le test de compatibilité. Cependant, le cas de la modification du processus global n'est pas abordé. Une évolution du processus global va engendrer des incompatibilités avec toutes les interfaces fournies de tous les partenaires abonnés. Dans ce cas, la résolution des incompatibilités est complexe et coûteuse à mettre en œuvre car elle suppose la maintenance des liens d'interactions entre tous les partenaires. Cette proposition est aussi limitée aux compositions de services selon le modèle d'orchestration permettant une coordination globale.

Dans [119], un moteur de découverte de services (*IPSI-PF : IPSI-Process Finder*) est proposé pour trouver les processus métiers, définis dans les interfaces comportementales des services, qui soient compatibles avec la requête exprimée par le client par son interface requise. C'est une extension du standard UDDI. L'extension proposée utilise les descriptions en BPEL des services fournisseurs, puis effectue des calculs de conformité⁵ avec l'interface requise par le client. Il s'agit de trouver des services dont les interfaces fournies sont compatibles avec l'interface requise. La requête initiale du service client se décompose en deux sous-requêtes :

- la première permet de trouver, entre autres, les adresses des services partenaires attendus dont les interfaces structurelles (descriptions des opérations dans le format WSDL) sont compatibles avec les descriptions des opérations attendues par le client ;
- la deuxième retourne les partenaires dont le comportement (descriptions des enchaînements des opérations en BPEL) est compatible avec le comportement attendu par le client.

⁵Le formalisme adopté par les auteurs est Workflow Net [105].

D'autres auteurs proposent d'étendre la description des services abonnés à un annuaire UDDI avec des pages bleues qui contiennent la définition des processus [87]. Cependant, aucun mécanisme de découverte de service à base de description de processus n'est mis en avant.

Les propositions ci-avant s'appliquent dans un contexte où les services clients découvrent à l'exécution les services fournisseurs avec qui ils vont interagir. Avant chaque exécution, le client reçoit une liste des services dont les interfaces fournies sont conformes avec son interface requise. Ceci assure que le client interagit toujours avec des services dont les interfaces fournies sont compatibles avec celle qu'il requiert. Si l'interface d'un service évolue, il ne sera pas sélectionnée dans la phase de découverte.

2.3 Changements des interfaces

Dans cette section, les approches présentées traitent des incompatibilités entre les services lorsque les définitions des interfaces fournies changent. En effet, un changement de l'interface d'un service engendre des incompatibilités avec l'interface requise par le client qui l'utilise. Dans la section 2.3.1, des approches suggèrent que les changements des interfaces sont regroupés en patrons de changements dans la perspective de résoudre les incompatibilités associées à l'aide de patrons d'adaptation. Ensuite, dans la section 2.3.2, les approches présentées traitent du cas où le changement dans la définition de l'interface d'un service survient au moment où des instances de conversations entre les clients et ce service sont en cours d'exécution. Enfin, la section 2.3.3 présentent les travaux qui s'appuient sur de la gestion des versions d'une interface comme moyen de prévention des incompatibilités lorsque la définition de l'interface d'un service change.

2.3.1 Patrons de changement

L'idée de la définition des patrons de changement des interfaces comportementales vise à orienter les concepteurs lorsqu'ils changent la définition de l'interface d'un service. Le concepteur doit également propager les changements aux applications clientes qui utilisent le service, en suivant les indications décrites dans des patrons d'adaptations. Il s'agit de prévenir les incohérences et les incompatibilités entre les services suite à ces changements. Dans [114], les auteurs ont identifié 17 patrons de changement des processus métiers. Selon chaque patron de changement, ils définissent un ensemble de règles à vérifier pour propager les changements du modèle d'un processus vers les autres modèles des processus des partenaires. Les règles de propagation des changements sont décrites dans

des *patrons d'adaptation*. La propagation des évolutions aux instances de processus en cours d'exécution est gérée par le concepteur en définissant des zones de changement où sont transférées les instances de l'ancienne définition du processus vers la nouvelle définition du processus. Dans le cas où les instances ne peuvent pas être transférées, elles continuent leur exécution selon l'ancienne définition du modèle du processus.

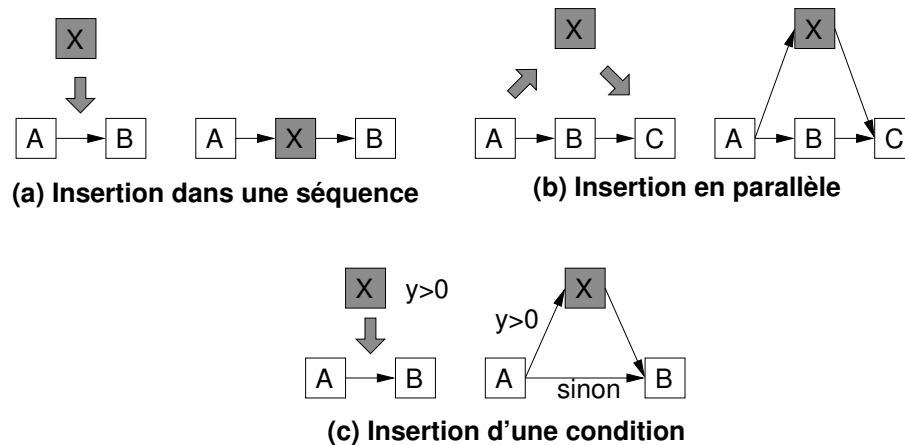


FIG. 2.6 – Patron d'insertion d'un fragment de processus

La figure 2.6 illustre le patron d'insertion qui présente les différentes manières d'insérer un fragment de processus dans le modèle d'un processus. Trois situations d'insertion sont identifiées :

- dans un séquence (voir (a)) : un fragment de processus X est introduit entre deux fragments de processus A et B. Une nouvelle séquence du processus est générée où A précède X et X précède B,
- parallèle (voir (b)) : X est introduit entre les fragments A et C et X s'exécute en parallèle avec B,
- condition (voir (c)) : X est introduit entre les fragments A et B mais X n'est exécuté que si une condition est vérifiée.

Cependant, l'utilisation des patrons d'adaptation qui décrivent la manière de propager des changements est conditionnée par l'hypothèse selon laquelle les modèles des processus métiers sont pris dans un contexte intra-organisationnel. En d'autres termes, lorsqu'un concepteur décide de changer le modèle de processus d'un partenaire, il a également la possibilité d'effectuer les changements nécessaires dans les modèles des autres processus partenaires affectés par ces changements. Ceci ne s'applique pas dans le contexte du web car le service qui évolue n'a aucun pouvoir sur les clients qui l'utilisent. De ce fait, il est impossible au concepteur d'un service web dont l'interface fournie évolue de propager ses changements aux applications clientes qui utilisent ce service.

Dans [95], les auteurs présentent le canevas *DYCHOR* (*DYnamic CHOReographies*) qui exploite la sémantique des changements de comportements dans les interfaces privées et publiques d'un service pour les propager automatiquement aux interfaces de ses partenaires. L'interface privée d'un service est une interface qui décrit les opérations internes qui ne sont pas visibles aux clients ainsi que les opérations de communication qui peuvent être sollicitées par les applications clientes. Toutefois, l'interface publique ne décrit que les opérations de communication. La propagation des changements n'est effectuée que si la conformité n'est plus garantie entre la nouvelle interface fournie et celle requise par le partenaire. Les auteurs utilisent des automates annotés pour formaliser le comportement des services. Les annotations sont des expressions logiques qui expriment des conditions qui doivent être vérifiées pour effectuer des envois de messages ou pour attendre des réceptions de messages. Les cas de changement traités sont ceux de la suppression et de l'ajout de séquences d'envoi et de réception de messages dans les interfaces. La définition de l'interface comportementale du service est traduite en un automate dans la perspective d'être comparée à l'automate qui représente la nouvelle définition de l'interface du service. Les changements entre les deux définitions de l'interface du service sont détectés au moyen d'un calcul des différences entre les deux automates qui les représentent. Les changements dans la définition de l'interface comportementale du service peuvent induire ou non des incompatibilités avec les partenaires qui interagissent avec le service. Afin de distinguer les situations où les changements induisent des incompatibilités, des situations où les changements n'induisent pas d'incompatibilités, les notions d'invariant et de variant sont introduites où :

- l'invariant désigne le fait que la nouvelle définition de l'interface publique fournie par le service reste compatible avec celles requises par ses clients. Dans ce cas, le changement dans l'interface publique n'affecte pas la compatibilité avec les interfaces publiques des partenaires,
- le variant désigne le fait que la nouvelle définition de l'interface publique n'est plus compatible avec celles requises par ses partenaires. La nouvelle interface publique n'est plus compatible avec celles des partenaires.

Lorsque la nouvelle définition de l'interface publique n'est plus compatible avec celles requises par ses partenaires, le changement est propagé aux interfaces publiques des autres partenaires. Les changements dans l'interface publique du partenaire sont ensuite propagés à son interface privée. Cette propagation est manuelle et nécessite l'intervention des développeurs. Cependant, les propagations des changements aux interfaces publiques et privées des partenaires ne sont possibles que si le concepteur du service dont la définition de l'interface fournie a évolué, a le droit de modifier les définitions des interfaces des autres partenaires.

2.3.2 Évolution des instances de processus en cours d'exécution

Le modèle de processus décrit dans l'interface comportementale d'un service représente l'enchaînement des interactions entre le service et ses clients. Une conversation initiée par un client avec le service est aussi une instance du modèle de processus représenté dans l'interface fournie du service. Lorsque la définition de l'interface fournie du service change vers une nouvelle définition de son interface, les conversations, initiées par les clients, ne sont pas toutes terminées. Ces conversations engendrent des incompatibilités avec la nouvelle définition de l'interface fournie. Dans [17, 94], les auteurs s'intéressent aux changements qui surviennent dans la définition d'un modèle de processus au moment où une instance de celui-ci est en cours d'exécution. Des zones de changements sont définies pour permettre la propagation des changements aux instances du processus en cours d'exécution dans la perspective de prévenir les incompatibilités. Une zone de changements est une partie du modèle du processus où des changements ont été opérés par le concepteur. A cet effet, les auteurs proposent un langage, WFML (*WorkFlow Modification Language*), qui offre des fonctions primitives pour définir un nouveau modèle du processus. Ce langage permet également à des instances de l'ancienne version du processus, qui sont en cours d'exécution, de continuer leurs exécutions selon le nouveau schéma du processus. Différentes politiques de migration des processus en cours d'exécution vers le nouveau schéma sont présentées. Elles sont classées en trois grandes catégories :

- l'annulation : les anciennes instances sont annulées ;
- la terminaison : les dernières instances de l'ancienne version du processus sont exécutées jusqu'à la fin ;
- la progression : les deux versions du processus coexistent et les instances de l'ancienne version du processus migrent vers le nouveau schéma du processus.

Toutefois, le cas d'une opération en cours d'exécution, mais qui a été supprimée dans la nouvelle définition du modèle de processus, n'a pas été abordé. Les instances de processus qui continuent de considérer une opération supprimée ne pourront pas aboutir car le résultat de l'opération ne peut plus être produit.

La figure 2.7 illustre le modèle du processus qui représente les enchaînements des opérations (voir *Modèle du processus*). Ce modèle est instancié en plusieurs instances du processus (voir *Instance l1* et *Instance l2*). Chaque instance du processus a un état d'avancement à un instant donné. L'état d'avancement d'une instance de processus en cours d'exécution est représenté par des fragments de processus en cours d'exécution (voir *o*) ou par des fragments de processus terminés (voir *•*). Une évolution de la définition du modèle du processus doit être, si possible, propagée à toutes les instances en cours d'exécution.

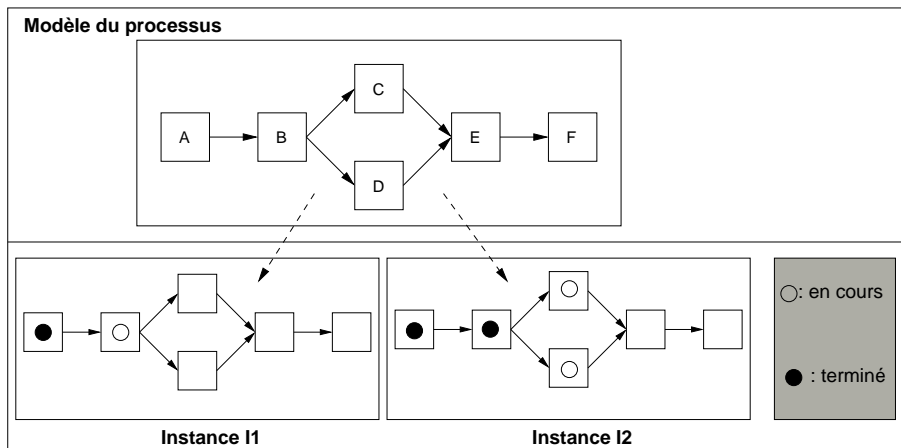


FIG. 2.7 – Modèle d'un processus et instances du modèle du processus

Dans l'exemple de la figure 2.7, supposons que le concepteur définit un nouveau modèle du processus avec une zone de changement dans le modèle du processus au niveau des fragments E et F. Le nouveau modèle est tel que le fragment F est désormais avant E. Les états d'avancement des instances du modèle du processus I1 et I2 indiquent que les activités définies dans les zones de changements ne sont pas encore en cours d'exécution. Dans ce cas, il est possible de propager le changement dans le modèle du processus à ces deux instances. Supposons maintenant que le concepteur définit un nouveau modèle du processus avec une zone de changement dans le modèle du processus au niveau des fragments C et D. L'état d'avancement de l'instance I1 du modèle du processus indique que les activités définies dans les zones de changements ne sont pas encore en cours d'exécution. Dans ce cas, il est possible de propager le changement décrit dans le nouveau modèle du processus à cette instance. Cependant, l'état d'avancement de l'instance I2 du modèle du processus indique que les activités définies dans les zones de changements sont en cours d'exécution. De ce fait, il est impossible de propager le changement décrit dans le nouveau modèle du processus à cette instance.

ADEPT est un moteur de modélisation, d'exécution et de contrôle de processus métiers [93]. Cet outil propose des opérations de modification du modèle du processus et vérifie si les changements peuvent être propagés aux instances du processus. Dans le cas contraire, une exception est levée et le changement n'est pas appliqué. L'exception qui est retournée permet uniquement de détecter l'incompatibilité. La résolution de cette dernière est à la charge du concepteur du processus.

Dans [30], les auteurs proposent, dans une problématique similaire liée à la propagation des changements aux instances de processus en cours d'exécution, la définition de régions de changement. Ces dernières identifient des parties du modèle du processus

concernées par cette évolution. Les actions délimitées par ces régions sont les seules à être transférées de l'ancienne version du processus à la nouvelle version du processus. Dans ce travaux, les auteurs parlent de transferts des changements pour désigner la propagation des changements dans le modèle du processus aux instances en cours d'exécution. Les processus sont modélisés par des réseaux de Petri [19, 79]. Dans chaque réseau de Petri qui modélise une instance du modèle de processus en cours d'exécution, les descriptions des régions sont modifiées pour obtenir de nouvelles régions. L'idée est d'effectuer le transfert des régions de changement des instances en cours d'exécution vers les nouvelles régions en spécifiant un nouveau marquage. Le marquage des nouvelles régions prend en comptes les actions en cours d'exécution dans les régions de changement. Les actions supprimées sont elles aussi prises en compte dans les définitions des nouvelles régions. Le cas extrême, dans la définition des régions de changement, est que le changement concerne tout le processus. Les instances de l'ancienne définition du modèle processus ne peuvent plus être transférées vers le nouveau modèle du processus. Cependant, aucun mécanisme de détection automatique des régions de changement n'est présenté. C'est le concepteur qui le désigne au moment de la définition du nouveau modèle processus.

Dans [110], les auteurs proposent d'introduire la notion d'héritage (spécialisation / généralisation), connue dans le domaine du génie logiciel, et de l'adapter à la modélisation des processus métiers. Par analogie, les classes correspondent aux définitions des modèles de processus métiers et les objets correspondent aux instances du modèle du processus métier. Chaque modification du modèle du processus génère un sous-modèle du modèle initial. Un graphe sous forme d'un arbre d'héritage de modèles de processus est ainsi obtenu, et dans lequel un modèle de processus hérite de la structure et du comportement des modèles dont il est sous-modèle. Ainsi, les nouvelles définitions du modèle du processus héritent de la structure et du comportement de l'ancienne définition du modèle du processus. En d'autres termes, le *processus fils* simule le comportement du *processus père*. Ce mécanisme impose d'avoir des modèles de processus incrémentaux qui permettent de maintenir plusieurs instances de différentes définitions d'un même modèle de processus. Cependant, cette approche n'est pas adaptée aux évolutions de la définition d'un modèle de processus suite à des suppressions et/ou à des modification d'opérations. En effet, si une opération est supprimée dans la définition d'un modèle de processus, le modèle de processus obtenu n'inclus pas le comportement du modèle dont il est dérivé.

Les études présentées ci-avant traitent des évolutions d'instances de modèle de processus en cours d'exécution, ne considèrent que des processus d'orchestration où les instances de processus sont contrôlées par une seule entité. Les interactions avec d'autres partenaires ne sont pas étudiées, ce qui en limite le champs d'application à des processus propriétaires au sein d'une même organisation. Dans un contexte inter-organisationnel, le transfert des

instances d'un modèle de processus en cours d'exécution vers un nouveau modèle ne peut se faire sans propagation des changements aux autres partenaires du processus qui continuent de considérer le modèle du processus avant son évolution.

2.3.3 Versions d'une interface fournie

Afin de maintenir la compatibilité des interfaces fournies et requises entre les différents acteurs dans une composition de services, le recours au maintien des versions d'interfaces (qui décrivent des processus) est une des pistes de recherche pour y parvenir. Une version de l'interface d'un service est une nouvelle définition de l'interface du même service qui est obtenue suite à des changements effectués par le concepteur dans l'ancienne définition de l'interface du service. Ainsi, les évolutions des interfaces fournies génèrent de nouvelles versions de ces interfaces qui coexistent et qui sont maintenues. L'objectif du maintien des versions d'une interface fournie est de garantir la conformité entre les interfaces requises qui sont compatibles avec les versions antérieures de l'interface fournie du service. Dans cette section, nous présentons les approches qui traitent de la gestion des versions d'une interface selon ses deux aspects : structurel et comportemental.

Interface structurelle :

Dans [115], les auteurs s'intéressent aux évolutions qui touchent les services dans une composition de services et proposent une approche à base de gestion des versions. Les changements au niveau de la mise en œuvre d'un service, tel que l'optimisation des algorithmes des opérations fournies, n'ont pas d'impact sur la définition des interfaces des services. Les signatures des opérations restent les mêmes. Dans ce cas le déploiement de la nouvelle version du service et le retrait de l'ancienne version est complètement transparents aux clients. La re-direction des messages envoyés par le client vers la nouvelle version est gérée par un annuaire de versions qui garantit la transparence aux clients. Si l'évolution vers une nouvelle version d'interface ne génère pas d'incompatibilités, alors elle est déployée en parallèle avec les précédentes versions. Les évolutions étudiées ne concernent que les interfaces structurelles et les nouvelles versions d'une interface sont définies suite à un ajout de nouvelles fonctionnalités.

BEA Systems [45] propose l'outil *BEA WebLogic Workshop* pour la gestion des versions multiples d'un service web. L'outil se limite aux définitions des opérations offertes par le service dans le fichier WSDL et ne prend pas en compte la description de l'interface comportementale du service. Les évolutions vers de nouvelles versions sont faites uniquement par des ajouts d'opérations ou d'attributs. Les aspects des évolutions com-

portementales liés aux suppressions et/ou aux modifications des opérations ne sont pas considérés.

Dans [65, 64], les auteurs proposent d'étudier les versions de l'interface structurelle d'un service. Chaque nouvelle version est obtenue par des changements des signatures des opérations et/ou par des ajouts de nouvelles opérations dans la précédente version de l'interface du service. Ce dernier est déployé selon la définition de son interface la plus récente. Les auteurs parlent de l'interface actuelle du service. Les clients continuent d'interagir avec le service selon les versions antérieures de l'interface fournie par le service engendrant ainsi des incompatibilités. Les auteurs proposent des adaptateurs pour garantir la conformité entre les clients et le service qui est déployé selon la définition de son interface actuelle. Les adaptateurs sont définis entre pour garantir la conformité deux versions consécutives de l'interface fournie. Ainsi la conformité entre tous les clients est le service selon sa version actuelle est maintenue par une chaîne d'adaptateurs.

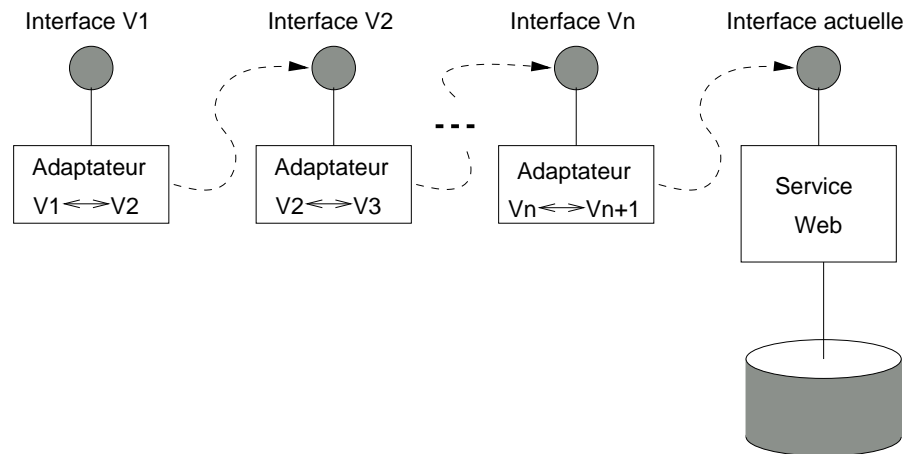


FIG. 2.8 – Chaîne d'adaptateurs après n versions

Dans la figure 2.8, l'adaptateur $V1 \leftrightarrow V2$ est défini entre deux versions consécutives de l'interface fournie du service (voir la flèche en pointillés entre l'adaptateur et l'interface $V2$). La version de l'interface $V2$ est plus récente que la version $V1$. Les clients qui interagissent avec le service selon la version $V1$ de l'interface du service envoient leurs messages tels que décrits dans cette interface. L'adaptateur reçoit ces messages de manière transparente au client. Ensuite, l'adaptateur transforme les structures des messages reçus vers des structures de messages qui sont compatibles avec celles définies dans l'interface du service selon la version $V2$. L'adaptateur $V1 \leftrightarrow V2$ envoie les nouveaux messages ainsi obtenus vers l'adaptateur $V2 \leftrightarrow V3$ qui expose l'interface $V2$. Après traitement des messages reçus, l'adaptateur $V2 \leftrightarrow V3$ retourne les messages résultats à l'adaptateur $V1 \leftrightarrow V2$. Ce dernier transforme les structures des messages reçus, telles que définies dans la version $V2$ de

l'interface du service, en structures des messages, telles que définies dans la version V1 de l'interface du même service. Les messages résultats ainsi obtenus sont envoyés par l'adaptateur vers les clients selon des structures identiques aux structures des messages définies dans les interfaces requises de ces clients.

Ainsi, pour un client qui continue d'interagir avec le service selon la version V1 de son interface fournie, ces messages transitent à travers les n adaptateurs et arrivent jusqu'à la version actuelle du service (et *vice-versa* pour les messages de réponse). Pour réduire ces re-directions, certains adaptateurs sont définis entre une version n et une autre version plus récente $n+k$ (où $k > 1$). En d'autres termes, l'adaptateur ne concerne pas que deux versions consécutives de l'interface du service.

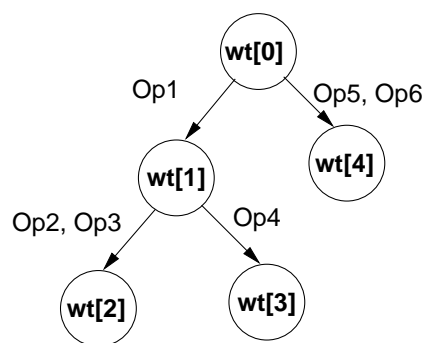
Cependant, les évolutions de l'interface fournie d'un service traitées ne concernent que l'aspect structurel relatif à la définition des opérations (modification des structures de paramètres en entrée et en sortie). La définition d'une nouvelle interface par un ajout d'opération est également traité. Lorsqu'une opération est supprimée, les auteurs supposent que la fonctionnalité qu'elle permet de réaliser sur des messages en entrée et les résultats obtenus en sortie peut être reproduite en combinant l'utilisation des autres opérations restantes. Cette hypothèse n'est pas toujours vraie dans la réalité. Par exemple, supposons que dans une interface deux opérations de paiement par transfert et par chèque soient définies. Lorsque l'opération de paiement par transfert est supprimée, le traitement qu'elle exécute ne peut pas être reproduit par l'opération de paiement par chèque. En effet, la structure des messages n'est pas la même.

Dans [63], les auteurs proposent une approche basée sur l'envoi d'un message qui alerte les clients abonnés à un service, des nouvelles modifications dans l'interface du service. L'approche proposée est un annuaire de services SOMR (*Service-Oriented Monitoring Registry* semblable à l'annuaire UDDI) auquel sont également abonnés des clients. Chaque client abonné à un service reçoit des messages d'alerte à chaque fois que la description de l'interface du service évolue. Chaque alerte est soit une nouvelle description du service (par exemple, le nouveau fichier WSDL) soit une information sur la non disponibilité du service. Après avoir reçu une alerte, les développeurs du client doivent adapter les interactions du client avec la nouvelle interface dans la perspective d'éviter les incompatibilités.

Dans cette approche, le service qui évolue maintient une seule version et la résolution des incompatibilités à chaque évolution vers une nouvelle version est à la charge des développeurs des applications clients. Cette solution ne traite pas des aspects comportementaux des services.

Interface comportementale :

Dans une démarche similaire de maintien de versions multiples d'interfaces fournies, l'aspect comportemental lié à la gestion des versions des modèles de processus a été étudiée dans [71]. Les auteurs proposent de maintenir les versions antérieures des modèles de processus pour permettre aux instances de processus, qui ne peuvent pas migrer vers la nouvelle version, de continuer leurs exécutions selon les précédentes versions. Les versions sont structurées en arbre, où une nouvelle version est ajoutée par dérivation d'une ancienne version selon une ou plusieurs opérations de modification. En d'autres termes, la nouvelle version est obtenue à partir de l'ancienne version en effectuant des changements dans cette ancienne version jusqu'à obtenir la description de la nouvelle version. L'exemple de la figure 2.9 illustre un arbre de versions du modèle de processus `wt`. Le processus `wt[1]` a été dérivé du processus `wt[0]` en appliquant l'opération `Op1` à `wt[0]`, et ainsi de suite pour les autres versions du processus. Des conditions présentées sous formes d'invariants doivent être vérifiées avant et après l'évolution du modèle vers une nouvelle version. Ces invariants sont inspirés des travaux sur les évolutions des schémas de bases de données. Le transfert des instances des anciennes versions vers la nouvelle version est possible sous la condition que l'état d'avancement des instances en cours d'exécution ne concernent pas encore les parties du modèle du processus qui ont subi des modifications. Par exemple, si l'instance d'un modèle de processus est arrivée dans un état d'avancement où une opération supprimée est en cours d'exécution, le transfert de cette instance vers la nouvelle version du modèle du processus n'est pas possible. Dans ce cas, l'instance doit continuer son exécution selon l'ancienne version du modèle du processus.

FIG. 2.9 – Arbre de versions du modèle de processus `wt` [71]

Dans un contexte intra-organisationnel, la migration des instances en cours d'exécution d'un modèle de processus vers d'autres versions du modèle de processus peut être envisagée et réalisée sous certaines conditions. Cependant, si le modèle du processus est

partagé par différentes organisations qui intervient dans une composition de services, la migration des instances du modèle du processus ne peut pas être réalisée. En effet, les modifications dans le modèle ne peuvent pas être propagées aux autres partenaires. De plus, la gestion et le maintien de plusieurs versions en parallèle d'un modèle de processus peuvent être contraints par des facteurs externes auxquels l'organisation doit se soumettre. En d'autres termes, l'organisation ne doit pas maintenir plusieurs versions d'un modèle de processus car elle n'en est pas autorisée. Par exemple, dans une version d'interface fournie, le processus de vente d'article en ligne n'impose pas de signature des clauses de conditions de vente par ses clients. Cependant, suite à une nouvelle loi qui impose la signature des conditions de vente, les anciennes versions du processus ne peuvent plus être exécutées car elles ne respectent pas cette nouvelle loi.

Dans [109], les auteurs traitent des changements des modèles de processus. Les travaux présentés suggèrent la modélisation de processus génériques qui peuvent se spécialiser pour en déduire de nouveaux modèles de processus. Chaque nouveau modèle de processus peut être vu comme une évolution suite à une extension du modèle processus générique. Les cas d'évolution traités sont dus à des extensions par :

- ajout d'opération (ajout d'une option, ajout d'opérations en parallèles, ajout d'une opération dans une séquence)
- remplacement d'une opération (ou d'un sous processus) par une autre opération (ou sous-processus).
- définition de l'ordre des opérations dans une séquence.

Dans l'approche ci-avant, les auteurs suggèrent la possibilité d'avoir plusieurs modèles de processus obtenus à partir d'un même modèle de processus global. Ainsi, dans le cas où le modèle du processus évolue vers un nouveau modèle, les partenaires qui sont compatibles avec l'ancien modèle du processus restent compatibles avec le nouveau modèle du processus. En effet, dans les extensions de modèle du processus, les nouvelles instances de processus incluent le comportement du processus générique. Ceci revient à dire que le nouveau comportement décrit dans le nouveau modèle du processus simule le comportement décrit dans le modèle générique du processus. Un des inconvénients de l'approche est qu'elle admet l'existence de plusieurs versions, en parallèle, d'un même processus métier. Cette solution ne peut pas être mise en œuvre dans le cas d'évolutions imposées par des contraintes externes où le maintien de plusieurs versions n'est pas permis. Une autre restriction de cette approche est qu'un des cas d'évolution du modèle de processus non pris en compte est le cas de la suppression d'une opération. En effet, les auteurs traitent des

extensions de processus par des ajouts d'opérations mais ne traitent pas des réductions de processus par des suppressions d'opérations.

Dans [62], les auteurs proposent un système de gestion de processus qui prend en compte les évolutions des modèles de processus et de leurs instances. La migration des instances vers un nouveau modèle est réalisée de manière soit automatique soit manuelle. Afin de garantir la conformité des instances qui migrent vers le nouveau modèle de processus, un retour en arrière de certaines activités peut être envisagé. Les modèles de processus sont décrits à l'aide de diagrammes de classes. L'aspect comportemental des processus est modélisé à l'aide de diagrammes d'états et de règles qui fixent les contraintes d'exécution des opérations. Lorsque les descriptions des opérations dans un modèle de processus changent, de nouvelles versions du modèle sont définies à partir de l'ancienne version. Un mécanisme de fusion de deux ou plusieurs anciennes versions vers une nouvelle version est proposé mais non détaillé. Les évolutions traitées sont applicables sous certaines conditions relatives aux états des instances en cours d'exécution. L'objectif de cette démarche est de prévenir les incompatibilités qui peuvent survenir pour les instances de processus en cours d'exécution. Par exemple, la suppression d'une opération est effectuée si aucune instance n'est en cours d'exécution de cette activité.

Cependant, dans cette approche, les changements traités ne prennent pas en compte d'autres modifications du modèle du processus tel que le changement de l'ordre d'exécution des activités dans une séquence. La propagation des changements aux instances en cours d'exécution est manuelle et elle est possible que dans ce contexte d'intra-organisation.

Dans [25], les auteurs proposent, dans un contexte intra-organisationnel basé sur des architectures orientées services, des outils et des modèles pour le déploiement des services et la génération des clients conformes aux interfaces des services. Cette solution est envisagée dans le cadre d'une seule organisation où les différents départements offrent des services qui sont reliés entre eux. Les développeurs ont accès aux services fournis aussi bien qu'aux applications clientes qui les utilisent. Si une nouvelle version du service est déployée, alors l'ancienne version est supprimée. L'outil permet au développeur de naviguer dans les services existants afin de trouver un ou plusieurs services qui répondent à certains besoins d'utilisation. Le contexte de cette approche est très restreint. En effet, il n'est pas applicable dans le cadre du web où les services clients ne sont accessibles que par leurs détenteurs. Il est de ce fait impossible pour un service fournisseur de définir de nouvelles applications clientes et partenaires qui sollicitent son interface fournie.

2.4 Synthèse

Nous présentons maintenant une synthèse comparative des approches que nous venons de décrire. Nous nous appuyons pour cela sur un ensemble de critères de comparaison qui sont issus de la problématique étudiée dans cette thèse et de ses objectifs.

Un premier critère oppose les approches selon qu'elles traitent ou pas, des deux dimensions structurelle et comportementale des interfaces (voir section 2.5.1). Puis, d'autres critères portent sur la détection des incompatibilités (section 2.5.2) et enfin un dernier ensemble de critères permet de comparer les approches selon le point de vue de la résolution de ces incompatibilités (section 2.5.3). Nous donnons dans la section 2.5.4 des tableaux résumant la discussion.

2.4.1 Dimensions structurelle et comportementale

Dans la problématique de détection et de résolution des incompatibilités entre les interfaces des services, deux dimensions complémentaires des interfaces doivent être prises en compte : i) la dimension structurelle et ii) la dimension comportementale. Les travaux présentés dans l'état de l'art se concentrent sur l'une ou l'autre de ces deux dimension.

Dimension structurelle :

La détection et la résolution des incompatibilités sur le plan structurel se ramène essentiellement à un problème de réconciliation entre les types des messages décrits dans les interfaces fournies et requises. Il existe sur ce sujet de très nombreuses études [92, 5, 9, 84, 29] et plusieurs systèmes proposant des solutions certes partielles, sont commercialisés (par exemple Microsoft's BizTalk Mapper). Des solutions basées sur la transformation des messages envoyés et reçus sont proposées. Les descriptions des messages échangés par un service sont fournies dans des fichiers WSDL. D'autres travaux proposent l'adaptation des interfaces des services web selon cette dimension structurelle [9, 29].

Dimension comportementale :

Les tests de conformité des interfaces sur le plan comportemental ont pour objectif de vérifier si les interactions, entre un service et un client, décrites dans l'interface fournie et dans l'interface requise n'engendrent pas d'incompatibilités. Afin d'effectuer des tests sur la conformité des interfaces sur le plan comportemental, les solutions proposées permettent de détecter une incompatibilité entre deux interfaces mais ne les détectent pas toutes [23]. Les mécanismes de résolution proposés dans ce cas sont basés sur une intervention humaine où le concepteur modifie les interfaces et réitère les tests de conformité jusqu'à obtenir des interfaces conformes.

2.4.2 Détection

Les approches proposées pour la détection des incompatibilités sont comparées entre elles sur la base d'un critère lié au moment où les tests de compatibilités entre les interfaces sont effectués. En effet, les tests de compatibilités entre les interfaces peuvent être réalisés à la conception et/ou à l'exécution des services. Les mécanismes de détection des incompatibilités doivent être automatiques.

Tests de compatibilité à la conception versus à l'exécution :

A la conception d'une composition de services ou d'un schéma de conversation entre les clients et les partenaires, la compatibilité entre les interfaces fournies et les interfaces requises doit être testée. Le concepteur de la composition apporte des modifications nécessaires dans le schéma des interactions et dans la mise en œuvre des applications clientes à chaque test négatif de la compatibilité. Une fois le test de compatibilité est positif, la composition de services est déployée comme un nouveau service. L'interface d'un service web peut évoluer et certaines évolutions induisent des incompatibilités aussi bien sur la plan structurel que comportemental avec les applications clientes et partenaires qui l'utilise. Les incompatibilités surviennent car la nouvelle version d'interface fournie ne simule pas le comportement de l'ancienne version de l'interface fournie. Sur le plan structurel, la structure des messages envoyés et reçus dans la nouvelle version n'est plus compatible avec la structure des messages envoyés et reçus dans l'ancienne version. Les tests de compatibilité à l'exécution suggère l'idée que les partenaires d'une composition de services ne sont pas connus à la conception. C'est au moment de l'exécution du service composé que les services partenaires sont choisis (à partir d'un annuaire de services) à condition qu'ils vérifient le test de compatibilité avec l'interface requise par la composition [103, 99].

Détection automatique des incompatibilités :

Dans la détection automatique des incompatibilités entre deux interfaces sur le plan comportemental, l'accent est mis sur le degré de simulation entre les interfaces. Des mesures de similarité entre graphes [18] sont réutilisées dans le contexte de simulation en prenant en compte la sémantique des enchaînements des opérations. Dans [83], les auteurs proposent une fonction récursive qui retourne une valeur maximisée qui définit le poids de la similarité entre deux interfaces comportementales. Dans leurs travaux, les auteurs se sont penchés sur une problématique de détection automatique de virus dans les programmes. En comparant le programme source original au programme potentiellement infecté par un virus. Cependant, le résultat retourné n'est qu'une valeur numérique de

mesure de similarité entre deux interfaces, qui n'indique pas pour autant la localisation précise des incompatibilités rencontrées. De plus les programmes traités sont autonomes et ne prennent pas en compte des aspects liés aux interactions avec d'autres programmes, comme c'est le cas dans les services web où tout est basé sur l'envoi et la réception de messages. En d'autres termes, des hypothèses de communication synchrone et de communication asynchrone des interactions sont à prendre en compte.

2.4.3 Résolution

La résolution des incompatibilités est un mécanisme qui n'est pas totalement automatisé. Les approches étudiées ont recours à l'intervention humaine pour résoudre les incompatibilités qui ne peuvent être résolues de manière automatique. D'autres approches proposent d'utiliser des mécanismes de gestion de versions des interfaces afin d'éviter les incompatibilités. Lorsque ces dernières se produisent, d'autres approches suggèrent de substituer le service dont l'interface n'est plus compatible par un nouveau service dont l'interface est compatible avec les interfaces requises par les partenaires. Le mécanisme de résolution doit être transparent pour les applications clientes qui utilisent le service.

Résolution automatique versus manuelle des incompatibilités :

La résolution des incompatibilités peut être faite à l'exécution par le concepteur du service qui évolue ou bien à la conception au niveau des applications clientes. Les concepteurs des applications clientes doivent effectuer les adaptations nécessaires de manière manuelle, afin de garantir la conformité de l'interface requise avec l'interface fournie. La détection des incompatibilités peut être exécutée de manière totalement automatique alors que la résolution demande parfois une intervention humaine. Dans ce cas, le concepteur doit revoir la mise en œuvre des applications clientes afin qu'elles restent compatibles avec la nouvelle version d'interface fournie du service qui a évolué. Du point de vue du service qui évolue, les solutions qui se basent sur des adaptateurs pour la résolution des incompatibilités, sont mises en œuvre de façon manuelle. Une fois les incompatibilités détectées, la résolution automatique de certains types d'incompatibilités n'est pas toujours possible. En effet, lorsque certaines évolutions de l'interface fournie impliquent que son environnement doit lui fournir certaines informations (par exemple, par ajout d'une opération de réception de coordonnées bancaires), la résolution automatique ne peut être envisagée car le médiateur ne peut pas créer une telle information. Toutefois, dans le cas où la résolution s'appuie sur la mémorisation des données, leur restructuration puis leur envoi et réception, alors le processus peut être automatisé [120, 29]. Par exemple, dans le cas où l'envoi d'un acquittement n'est plus nécessaire, cette information est stockée sans être

transmise. Pour les cas qui ne peuvent être automatiquement résolus, une intervention humaine est impérative, d'où l'intérêt de bien rapporter la localisation des incompatibilités. Dans ce cas une intervention du côté client n'est pas à exclure. Par exemple, dans le cas d'un ajout d'une opération qui sollicite des informations que seul le client peut envoyer, le concepteur de l'application cliente doit ajouter une opération d'envoi de ces informations.

Résolution par substitution de services versus par gestion de versions :

En cas d'incompatibilités entre l'interface fournie d'un service qui évolue et les interfaces requises de ses clients, les clients préfèrent substituer le service qui n'est plus compatible avec leurs interfaces requises par un autre service. Le service substitut offre les mêmes opérations que le service substitué et son interface fournie est compatible avec l'interface requise du client. Des approches proposent des annuaires de services qui offrent des mécanismes de découverte de services. Les services sélectionnés dans l'annuaire répondent aux besoins exprimés par les requêtes des clients qui optent pour ce mécanisme de résolution. Le client choisit un service qui substitue le service qui engendre des incompatibilités dans les conversations. Le service qui évolue peut décider de maintenir son ancienne version d'interface fournie et proposer en parallèle une nouvelle version. L'objectif du maintien et de la gestion des versions multiples des interfaces est d'assurer la compatibilité des interfaces requises des applications clientes et partenaires qui continuent de considérer les anciennes versions d'interfaces fournies dans leurs schémas d'interactions. Le maintien de versions multiples d'interface permet également de garantir des interactions cohérentes dans les instances de processus d'anciennes versions en cours d'exécution. Les instances de l'ancienne version du schéma de processus peuvent migrer vers le nouveau schéma du processus métier. Cependant, cette solution n'est pas toujours envisageable (par exemple, les changements sont imposés par l'environnement) et le maintien de plusieurs versions d'un service peut être très coûteux [109].

Résolution transparente aux clients :

Afin de garantir une résolution transparente des incompatibilités, l'accent est mis sur le fait que le service qui évolue est celui qui offre la solution pour la résolution des incompatibilités. La résolution est dans ce cas transparente à tous les clients du service. Une solution envisageable pour une résolution transparente des incompatibilités est l'introduction d'un médiateur entre le service fournisseur et ses clients. L'approche de médiation pour résoudre les incompatibilités introduit une notion d'intermédiaire dans la conversation entre services web [97, 5]. Le principe de la médiation est que tous les messages

échangés entre deux interlocuteurs passent par un service médiateur aussi nommé fournisseur virtuel. Ce médiateur a pour charge de résoudre les incompatibilités en transformant les messages échangés. Les transformations des messages garantissent que les structures des messages envoyés sont compatibles les structures des messages reçus et *vice-versa*. Toutefois, toutes les incompatibilités ne peuvent être résolues par médiation de la conversation. En effet, dans le cas où le changement est un ajout d'opération qui sollicite de l'information qui provienne de l'environnement du service (par exemple, les coordonnées bancaires d'un client), le médiateur ne peut pas créer cette information et la transmettre au service.

Evolution des instances d'un processus en cours d'exécution :

Concernant les instances de processus en cours d'exécution qui doivent s'adapter à la nouvelle version ([71, 110, 95]), les approches proposées suggèrent la propagation des changements du modèle de processus à ses instances en cours d'exécution. Ces approches s'appliquent dans des contextes intra-organisationnels, car le concepteur du nouveau modèle a la possibilité de modifier les applications partenaires qui interviennent dans le processus. Les instances d'un modèle de processus en cours d'exécution migrent vers le nouveau modèle de processus. Toutefois, la propagation des changements aux partenaires d'un processus n'est pas possible dans le cas où les applications partenaires sont propriétaires. En effet, le concepteur du processus qui évolue n'a pas le droit de modifier les définitions des applications partenaires pour leur propager les changements.

2.4.4 Tableaux récapitulatifs

Afin de détecter et de résoudre les incompatibilités entre les interfaces fournies et les interfaces requises, plusieurs travaux ont été proposés et chaque approche satisfait quelques uns des critères fixés présenté ci-avant. Les deux tableaux 2.1 et 2.2 résument l'ensemble des approches présentées qui traitent des incompatibilités entre les interfaces des services.

Le tableau 2.1 synthétise les critères pris en compte par les approches qui traitent des incompatibilités des conversations, engendrées suite à des évolutions des interfaces fournies. Les approches ([78, 63, 65, 115]) qui traitent des incompatibilités structurelles, considèrent les services web comme des ensembles d'opérations indépendantes entre elles et qui peuvent être exécutées dans n'importe quel ordre. La description de l'enchaînement des opérations n'est pas pris en compte. Cette définition d'un service web est très proche de celle d'un composant logiciel. La résolution des incompatibilités suite à évolutions des interfaces structurelles est inspirées des techniques de généralisation/spécialisation

Approche	1	2	3	4	5	6	7	8
[30]		X			X		X	
[17]		X	X			X	X	
[71]		X	X		X		X	X
[78]	X			X	X			
[109, 110]		X	X		X		X	X
[63]	X					X		
[119]		X		X	X			
DYCHOR [95]		X			X			
[65]	X				X		X	X
[114]		X	X			X	X	
[115]	X		X		X		X	X

- 1 : compatibilité de la structure
- 2 : compatibilité du comportement
- 3 : gestion de versions
- 4 : substitution de services
- 5 : détection et résolution automatique
- 6 : détection et résolution semi-automatique
- 7 : évolution des instances en cours d'exécution
- 8 : résolution transparente aux clients

TAB. 2.1 – Synthèse des approches avec prise en compte de l'évolution

connues des approches orientées objets. La détection et la résolution des incompatibilités sont automatiques. Cependant, les évolutions traitées sont limitées et ne concernent que les incréments des interfaces par des ajouts d'opérations ou bien par des ajouts des attributs à des opérations déjà existantes. Cette approche qui traite des évolutions structurelles des interfaces par un mécanisme de généralisation/spécialisation des interfaces, a également été adoptée par des approches qui traitent des évolutions comportementales des services web [109, 71]. L'idée consiste à considérer un schéma de processus comme une classe qui évolue de manière incrémentale en spécialisant la classe mère (processus père) en d'autres classes (processus fils) qui constituent de nouvelles versions du service.

La plupart des approches qui traitent des évolutions des services sur le plan comportemental, s'intéressent à la propagation des évolutions aux instances en cours d'exécution ([17, 71, 110, 114]). Cette démarche n'est cependant possible que dans un contexte où les services sont déployés dans la même entreprise. Dans le contexte du web où les services sont indépendants, une telle propagation des changements dans la définition des interfaces n'est pas faisable, car les applications clientes sont propriétaires et ne peuvent pas être modifiées par le service qu'elles sollicitent. Dans [115], les auteurs proposent de gérer les évolutions des interfaces des services web uniquement sur le plan structurel et ne considèrent que les ajouts de nouvelles opérations dans les nouvelles versions interfaces. Les évolutions traitées concernent des modifications des programmes des opérations en internes. En d'autres termes, il s'agit d'évolution pour optimiser les algorithmes des applications ou pour corriger des erreurs de programmation. Ces modifications n'ont pas d'incidences sur les signatures des opérations dans les interfaces mais impliquent le déploiement de nouvelles versions du service. Ces versions sont gérées de manière transparente aux clients et aux partenaires qui consomment les opérations offertes par ce service. Toutefois, cette approche ne peut pas s'appliquer dès lors que les modifications des opérations en internes ont un impact sur l'interface publique du service.

Le tableau 2.2 résume un ensemble des approches qui s'intéressent aux incompatibilités détectées lors de la conception d'une composition de services. Ces approches traitent de l'exécution d'une composition de services dont les partenaires ne sont pas connus à l'avance. Dans les approches [6, 88, 66, 80, 117], l'accent est mis sur la découverte de services qui sont compatibles d'un point de vue structurel ou comportemental avec la requête d'un client qui exprime son interface requise. La requête est généralement traitée par un moteur de découverte de services déployé sur un registre de services tel que UDDI. Certaines approches ([6, 88, 66]) proposent d'ajouter des descripteurs sémantiques en se basant sur des ontologies de domaines afin d'appliquer des opérations (par exemple, l'inférence) qui permettent de raffiner les requêtes. Le principal inconvénient de l'approche est qu'il n'existe pas une ontologie commune à tous les domaines. Les approches [13, 38, 9]

Approche	1	2	3	4	5	6
[6, 88]	X		X	X		
[1]	X		X		X	X
[66]	X		X		X	
[13, 12]		X		X		
[38]		X			X	
[80]	X		X	X		
[117]		X	X	X		
[9]	X				X	
[99]		X	X	X		
[101]	X	X	X		X	
[97, 5, 9, 29]	X	X			X	
[102]	X		X	X		X
[100, 82, 74]		X			X	

- 1 : Compatibilité de la structure
- 2 : Compatibilité du comportement
- 3 : Sustitution de services
- 4 : Détection et résolution automatique
- 5 : Détection et résolution semi-automatique
- 6 : Résolution transparente aux clients

TAB. 2.2 – Synthèse des approches sans prise en compte de l'évolution

visent la vérification de la conformité des interfaces au moment de la conception d'une composition de services. Dans le cas où la compatibilité n'est pas garantie directement entre l'interface fournie et l'interface requise, des adaptateurs sont introduits pour effectuer des transformations de messages afin de les rendre conformes [97, 5, 9, 29, 102]. Ces solutions ne prennent en compte que les aspects structuraux. Les approches présentées dans [100, 74] proposent des algorithmes de mesure de similarité comportementales entre les graphes. Les interfaces comportementales sont représentées par des automates en nombre d'états fini et les mesures de similarité retournent des valeurs qui indiquent le degré de similarité comportementale entre les deux automates comparés. Le mécanisme de résolution des incompatibilités entre les interfaces est manuel et à chaque modification de graphe, la mesure de similarité est appliquée de nouveau.

2.5 Conclusion

Dans l'état de l'art, nous avons présenté les différentes approches qui traitent du problème de conformité des interfaces entre deux partenaires qui interagissent dans une conversation. Pour garantir cette conformité, des idées ont été émises dans le contexte de découverte de services. L'accent est mis sur les façons de retrouver un ou plusieurs services dont les interfaces fournies répondent à une requête d'un service client qui est son interface requise. Les tests de conformité entre interface fournie et interface requise ne concernent que les aspects structuraux des interfaces. Des extensions aux aspects sémantiques par des définitions d'ontologies de domaine sur les opérations et les informations manipulées sont ajoutés pour enrichir les tests de conformité avec des mécanismes d'inférence. Les règles d'inférence utilisent la relation de généralisation/spécialisation qui relie les entités d'une ontologie. Les aspects comportementaux ne sont, cependant pas abordés. Un autre axe de recherche regroupe les approches qui étudient la conformité des interfaces suite à des changements sur le plan comportemental. Les mécanismes de détection des incompatibilités ne retournent que des réponses booléennes, vrai si la conformité est assurée, faux dans le cas contraire. Cette détection est complétée par une intervention humaine où le concepteur analyse visuellement les interfaces afin de comprendre l'origine de la première incompatibilité retrouvée par le test de conformité. Pour la résolution, les approches suggèrent que le service qui évolue peut répercuter ces changements sur tous ses partenaires et clients, chose qui ne peut pas être appliquée dans le contexte de services web.

Dans la deuxième partie de cette thèse, nous présentons le canevas ArchiMed qui est la proposition pour la détection et la résolution des incompatibilités entre les interfaces des services.

Deuxième partie

Proposition d'un canevas pour la
réconciliation de conversations par
génération automatique de médiateurs.

Chapitre 3

Modélisation des interfaces des services web et principes généraux de la solution

Sommaire

3.1	Introduction	69
3.2	Modélisation des interfaces des services	70
3.2.1	Modélisation du comportement	70
3.2.2	Modélisation de l'interface structurelle	72
3.3	Principes généraux de la proposition	73
3.3.1	Principe de détection des incompatibilités par détection des différences	73
3.3.2	Simulation des interfaces et maintien de la conformité	76
3.3.3	Principe de la résolution par médiation	78
3.4	Conclusion	79

3.1 Introduction

Dans cette deuxième partie de la thèse nous introduisons la proposition du canevas *ArchiMed* pour la réconciliation de conversations par génération automatique de médiateurs. La détection des incompatibilités entre les interfaces des services s'appuie sur des représentations formelles de ces interfaces dans la perspective d'y appliquer des tests de conformité et de simulation. Ainsi, avant de détailler notre proposition nous présentons dans ce chapitre, la modélisation des interfaces comportementales des services web sur laquelle se base notre proposition. Il s'agit d'une modélisation en automates déterministes avec un nombre fini d'états.

Ce choix de modélisation est justifié par ce qui suit :

- la modélisation en automates des comportements des systèmes est simple à comprendre. Elle a été largement utilisée dans des travaux antérieurs pour l’analyse des interfaces comportementales des services [13, 9, 83],
- tous les aspects comportementaux des interfaces des services peuvent être modélisés à l’aide des automates déterministes (séquence, alternative, parallélisme, synchronisation, etc.) [39, 111, 81, 11],
- les définitions et les résultats déjà démontrés tels que la simulation, la bisimulation et l’équivalence des automates peuvent être ré-exploités [20],
- il existe des transformations en automates des interfaces comportementales décrites dans d’autres modèles. Par exemple, la transformation d’une interface de BPEL vers un automate peut être réalisée grâce à un outil existant tel que WS-Engineer [33].

En partant de la théorie des automates, nous proposons une modélisation des interfaces comportementales et structurelles des services (voir la section 3.2). Un des résultats de la théorie des automates exploité dans notre approche et celui de la simulation des interfaces (voir la section 3.3.2). En partant de cette représentation en automates, des opérateurs pour la formalisation des incompatibilités sont introduits dans la section 4.2.1. Dans notre approche de résolution des incompatibilités, nous avons opté pour la médiation de la conversation (voir la section 5.4).

3.2 Modélisation des interfaces des services

Nous avons choisi d’exprimer l’interface comportementale et structurelle d’un service web par un système de transitions étiquetées (*Labelled Transition Systems*, LTS). Cette modélisation en automates est présentée dans les sections 3.2.1 et 3.2.2. Dans la section 3.3.2, nous introduisons des opérateurs sur les automates afin de modéliser les incompatibilités entre les interfaces.

3.2.1 Modélisation du comportement

Un LTS est un automate qui peut être représenté par un graphe orienté où les nœuds désignent les états possibles d’un système (avec un état initial et un ou plusieurs états finals) et les arcs désignent les transitions entre les états. Chaque arc est étiqueté par l’événement dont l’occurrence déclenche le passage de l’état origine de la transition vers l’état cible. Une condition (ou garde) sous forme d’expression booléenne peut être associée à une transition et, si cette condition est vérifiée l’état cible de la transition est atteint. Dans la modélisation des interfaces des services web, les transitions sont étiquetées par

les messages échangés entre les partenaires. Une transition est déclenchée à l'envoi ou à la réception d'un message et lorsque la condition associée, si elle est définie, est vérifiée. Un état désigne le fait que le processus est dans une phase donnée. Des opérations de manipulation de données peuvent également être associées à un état pour modifier les valeurs des variables du contexte d'une instance de processus [117, 89].

Dans sa définition formelle, un LTS est un tuple (S, L, T, s_0, F) où : S est un ensemble fini d'états, L est l'ensemble d'étiquettes (signatures des opérations et gardes), T est la fonction de transition ($T : S \times L \rightarrow S$), s_0 est l'état initial et F est l'ensemble des états finals telle que $F \subset S$. La fonction de transition T associe à un état source $s_1 \in S$ et à un libellé $l_1 \in L$ un état cible $s_2 \in S$. Dans cette modélisation, les transitions sont représentées par des tuples qui contiennent un état source, un libellé et un état cible.

Comme nous l'avons déjà souligné, l'envoi de messages est à la base des interactions entre des services. La réception d'un message se traduit par l'exécution d'une ou de plusieurs opérations par le service qui le reçoit. Les opérations mentionnées ici, dites de *communication*, déclenchent d'autres opérations dites *internes* implantées par des applications propriétaires. Nous notons par $>m$ (respectivement $<m$) l'envoi (respectivement la réception) du message m . Chaque conversation entre le service et un client génère une instance de l'automate qui identifie la conversation.

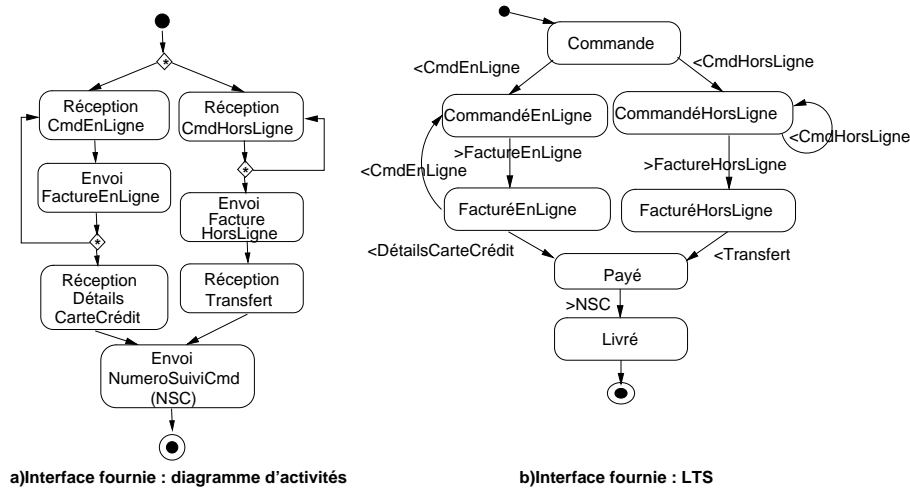


FIG. 3.1 – Modélisation en LTS de l'interface d'un service

La figure 3.1 (b) illustre la représentation en *LTS* du processus représenté en diagramme d'activités (voir la figure 3.1 (a)) de l'interface d'un service. Dans cet exemple, le service offre dans son interface la possibilité à ses clients de commander des articles en ligne ou hors ligne. Pour les commandes en ligne (respectivement hors ligne), la facturation se fait en ligne (respectivement hors ligne). Pour les commande en ligne, le service offre la

possibilité aux clients de modifier leurs commandes même après la facturation (voir, dans la figure 3.1 (a), le flux qui boucle sur l'activité de réception de commande en ligne après l'activité d'envoi de la facture en ligne et voir, dans la figure 3.1 (b), la transition étiquetée par `<CmdEnLigne` dont l'état source est `FacturéEnLigne` et l'état cible `CommandéEnLigne`). Pour les commandes hors ligne, les clients peuvent modifier leurs commandes tant que la facture n'a pas été envoyée par le service (voir, dans la figure 3.1 (a), le flux qui boucle sur l'activité de réception de commande hors ligne, et voir dans la figure 3.1 (b), la transition étiquetée par `<CmdEnLigne` dont l'état source est le même que l'état cible `CommandéHorsLigne`). Les factures des commandes en ligne (respectivement hors ligne) sont payées par les clients par carte de crédit (respectivement transfert). Une fois la facture payée, le service envoie au client un numéro de suivi de la commande (NSC) pour la livraison.

3.2.2 Modélisation de l'interface structurelle

La définition de l'interface structurelle d'un service web est liée à la définition des signatures des opérations que ce service offre aux clients. Comme dans une description WSDL d'un service, l'interface structurelle définit :

- les noms des opérations,
- les paramètres (en entrée et en sortie) des opérations ainsi que leurs types.

Dans la modélisation des interfaces par des automates, la description structurelle des opérations porte sur les libellés des transitions et sur la structure des messages envoyés et reçus. Chaque message reçu (respectivement envoyé) porte le nom d'une opération avec les valeurs des paramètres en entrée (respectivement en sortie). Chaque paramètre est d'un type particulier. Les types des données échangées sont définis dans des structures à part qui sont utilisées dans la description du LTS.

Dans notre proposition, des opérateurs de typage sont définis et permettent de retourner la nature des messages (en envoi ou en réception) ainsi que les éléments (ou partie) qui constitue les paramètres des opérations dans le message :

- *Polarité()* : indique si l'opération associée à une transition est un envoi de message (`>`) ou bien une réception de message (`<`).
- *Message()* : retourne le message d'une opération associée à une transition.
- *SousStructure()* : retourne une décomposition d'un message, envoyé ou reçu, en sous-structures qui le compose.
- *Garde()* : retourne l'expression de la garde d'une transition

D'autres opérateurs de manipulation des structures des messages seront introduites, plus loin, au fur et à mesure de l'introduction des principes de la détection et de la résolution des incompatibilités.

3.3 Principes généraux de la proposition

Nous considérons l'interface requise R d'un client conforme à l'interface fournie P d'un service. L'objectif de la détection est de déterminer si R est conforme à l'interface fournie P' d'un autre service (P' peut être issue de modifications effectuées sur P), et en cas de non conformité, d'en déterminer les raisons. Pour cela, une comparaison des deux interfaces P et P' est effectuée et vise à identifier toutes les incompatibilités entre elles (voir section 3.3.1). Dans la section 3.3.2 nous identifions les situations où les différences entre les interfaces P et P' ne conduisent à aucune incompatibilité. La section 3.3.3 discute de la résolution qu'il est possible d'effectuer sous certaines conditions.

3.3.1 Principe de détection des incompatibilités par détection des différences

Afin de vérifier si une interface fournie P' d'un service est conforme avec les interfaces requises des clients d'un autre service P , il suffit de s'assurer que l'interface fournie P' simule le comportement de l'interface P . Si l'interface P' ne simule pas le comportement de P , alors des incompatibilités avec les interfaces requises des clients du service P apparaissent. Notre objectif est de détecter toutes les incompatibilités. Afin de détecter ces incompatibilités entre les interfaces requises des clients et l'interface fournie P' , il faut détecter les différences entre l'interface fournie P et l'interface fournie P' qui font que P' ne simule pas P . En effet, il s'agit de trouver les différences, entre les deux interfaces fournies, qui sont à la source des incompatibilités entre l'interface fournie P' et les interfaces requises. Ces interfaces requises sont, rappelons-le, conformes à l'interface fournie P .

La figure 3.2 schématise les interactions entre un client service selon son interface fournie P et un autre service selon son interface fournie P' . Dans le schéma des interactions entre le client et le service P , les interfaces fournie et requise sont conformes. En effet, le client initie la conversation par l'envoi du message `CmdHorsLigne` qui est reçu par le service. Ce dernier offre au client la possibilité de modifier sa commande autant de fois qu'il le souhaite et ce, tant que la facture n'a pas été envoyée. Le client de son côté, exploite cette option en définissant un flux qui boucle sur l'envoi de commande pour modifier la commande en cours. Après la facturation puis le paiement par transfert, le client reçoit un numéro de suivi de commande `NSC` qui l'informe sur la livraison de sa commande. La conversation se termine sans incompatibilité. Toutefois, suite à la suppression du flux qui boucle sur la réception d'une commande hors ligne dans l'interface fournie P' d'un autre service, l'interface requise du client n'est plus conforme cette interface P' . Si le client désire modifier sa commande, il envoie de nouveau le message `CmdHorsLigne` et ce

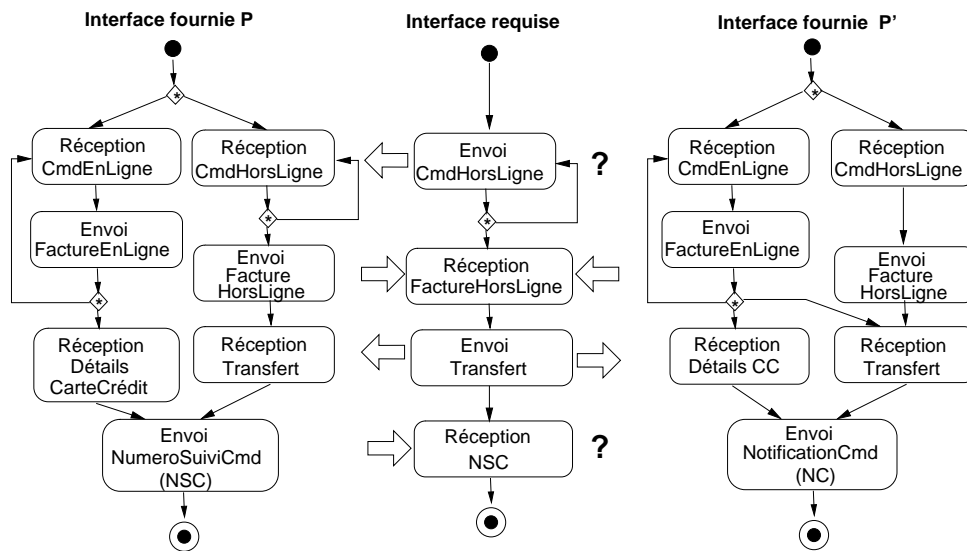


FIG. 3.2 – Exemple d'incompatibilités

message ne sera jamais reçu par le service P' . L'envoi du message contenant le numéro de suivi de commande NSC dans l'interface fournie P est modifié par un envoi de message de notification de commande NC dans l'interface fournie P' . La structure du message attendu par le client est différente de la structure du message envoyé par le service P' . Cette conversation échoue et se termine avec des incompatibilités.

Ainsi, pour détecter les incompatibilités entre l'interface requise du client et l'interface P' fournie d'un autre service, il suffit de détecter les différences (ajout, suppression et modification d'opérations) entre l'interface fournie P' et l'interface fournie P .

La détection des incompatibilités entre une interface requise et une interface fournie P' se traduit par la détection des différences entre les interfaces fournies qui induisent ces incompatibilités. Une abstraction de l'interface requise est faite pour ne considérer que les deux interfaces fournies P et P' . L'exemple de la figure 3.3 illustre les LTSs des deux interfaces fournies des services ainsi que les différences qui induisent des incompatibilités avec les interfaces requises des applications clientes. Les deux LTSs des deux interfaces P et P' traduisent les interfaces comportementales associées aux diagrammes d'activités donnés dans la figure 3.2.

Le résultat de la détection d'une incompatibilité élémentaire est un tuple (s, l, s', l', d) , où :

- s (respectivement s') est un état de P (respectivement P')
- l (respectivement l') est le libellé d'une transition sortante de s (respectivement s') qui peut être sans valeur (*nulle*).
- d est l'intitulé de l'incompatibilité élémentaire (ajout, suppression ou modification)

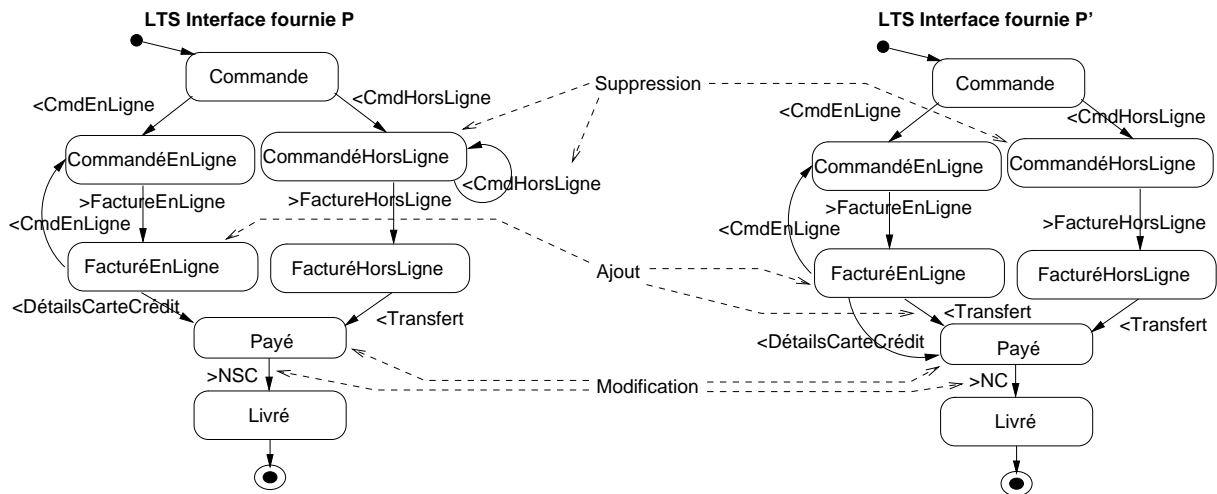


FIG. 3.3 – Détection des différences entre deux interfaces fournies

Le différence entre l'interface P' et l'interface P suite à la suppression de l'opération $\langle \text{CmdHorsLigne} \rangle$ est détectée au niveau des états CommandéHorsLigne des deux LTSs. La suppression de cette opération induit une incompatibilité car l'interface P' ne peut pas simuler le comportement de cette opération qui est offerte dans l'interface P (voir, dans la figure 3.3, les flèches en pointillés qui partent de **Suppression**). Le résultat de la détection de cette incompatibilité est donné par le tuple :

($\text{CommandéHorsLigne}, \langle \text{CmdHorsLigne} \rangle, \text{CommandéHorsLigne}, \text{nulle}, \text{'suppression'}$).

L'envoi du message NSC dans l'interface P est modifié par l'envoi du message NC dans l'interface P' . Cette différence dans les interfaces fournies est détectée en comparant les états Payé des deux LTSs P et P' (voir, dans la figure 3.3, les flèches en pointillés qui partent de **Modification**). Cette modification d'opération induit une incompatibilité car P' ne simule pas le comportement d'envoi du message NSC . Le résultat de la détection de cette incompatibilité est donné par le tuple :

($\text{Payé}, \text{'>NSC'}, \text{Payé}, \text{'>NC'}, \text{'modification'}$).

En comparant les deux états FacturéEnLigne des deux LTSs P et P' , l'ajout de l'opération de réception du message Transfert dans P' est détecté. Néanmoins, cet ajout n'induit pas d'incompatibilité car l'interface fournie P' offre cette opération en option au client qui peuvent ne pas envoyer ce message. De plus, P' simule le comportement de réception du message $\text{DétailsCarteCrédit}$. Ainsi, les différences qui n'induisent pas d'incompatibilités ne sont pas retournés comme résultats de la détection.

3.3.2 Simulation des interfaces et maintien de la conformité

La détection des incompatibilités entre les interfaces P et P' n'est pertinente que dans le cas où P' ne simule pas P .

Afin de vérifier si P' simule P il s'agit de parcourir les états et les transitions des deux LTSs de manière synchrone. A chaque paire d'états visitée, il faut vérifier si tous les libellés des transitions sortantes d'un état courant de P apparaissent dans les libellés des transitions sortantes de l'état courant dans P' . En d'autres termes, le service d'interface P' inclut le comportement du service d'interface P .

Ainsi, la définition de la simulation est donnée comme suit [100] :

Soient deux LTSs P and P' initialisés respectivement aux états s_0 et s_0' . P' simule P (noté $P' \preceq P$) si et seulement si :

- l'ensemble des libellés des transitions sortantes de s_0 est inclus ou égal à l'ensemble des libellés des transitions sortantes de l'état s_0' .
- pour chaque couple d'état (s_1, s_1') qui sont des états cibles des transitions sortantes de (s_0, s_0') dont les libellés sont identiques, alors l'automate P_1' (sous automate de P' initialisé en s_1') simule P_1 (sous automate de P initialisé en s_1).

Les situations qui nous intéressent sont celles où P' ne simule pas P . En effet, dans le cas où P' simule P (noté $P \preceq P'$) alors toute interface requise R d'un service client compatible avec P (noté $R \sim P$) reste compatible avec la nouvelle version d'interface P' (noté $R \sim P'$). Ce résultat est démontré ci-après. \bar{R} dénote l'interface contraire de R obtenue en transformant dans R les envois de messages en réceptions de messages et les réceptions de messages en envois de messages.

Hypothèses :

- (1) : P, P' des LTSs telle que $P \preceq P'$ { P' simule P . }
 (2) : R est le LTS d'une interface requise telle que $R \sim P$ { R compatible avec P }

Démonstration :

- (3) : (2) $\Rightarrow \bar{R} \preceq P$ { Selon la définition de l'interface contraire de R . }
 (4) : (3) \wedge (1) $\Rightarrow \bar{R} \preceq P'$ { Par transitivité de la relation de simulation }
 $\Rightarrow R \sim P'$

Conclusion :

$$R \sim P \wedge P \preceq P' \Rightarrow R \sim P'$$

Étant donné que $R \sim P$ alors $\bar{R} \preceq P$ (c'est-à-dire R est compatible avec P alors P simule \bar{R}) [13]. Par transitivité de la relation de pré-ordre de la simulation [20], il s'en suit que $\bar{R} \preceq P'$ (car $\bar{R} \preceq P$ et $P \preceq P'$). En conclusion, R est conforme à P' ($R \sim P'$). Ainsi, la détection et la résolution des incompatibilités ne sont nécessaires que lorsque P' ne simule pas P .

L'exemple de la figure 3.4 illustre deux interfaces fournies P et P' telle que P' simule P . En considérant la paire d'états initiale (Commande, Commande), les libellés de leurs transitions sortantes sont les mêmes (voir \langle CmdEnLigne). Le parcours synchrone des deux LTSs amène à visiter la paire d'états (CommandéEnLigne, CommandéEnLigne) dont les libellés des transitions sortantes respectives sont identiques (voir \rangle FactureEnLigne). La prochaine paire d'états à étudier est (FacturéEnLigne, FacturéEnLigne). A cette étape du parcours des deux LTSs, tous les libellés des transitions sortantes de l'état courant CommandéEnLigne de P sont inclus dans les libellés des transitions sortantes de l'état courant CommandéEnLigne de P' . A ce niveau du parcours, P' simule P car le libellé qui apparaît en plus dans les transitions sortantes de l'état courant de P' est une opération de réception de transfert qui est optionnelle. Le paiement par chèque est toujours défini dans P' et le paiement par transfert est un autre moyen de paiement qui peut ne pas être choisi par le client. Dans le test de simulation, une paire d'états est visitée au plus une seule fois. Ainsi, les libellés des transitions sortantes qui bouclent sur la paire d'états (CommandéEnLigne, CommandéEnLigne) ne seront pas considérés dans les prochaines étapes du parcours de P et P' . Ainsi l'interface P' simule P .

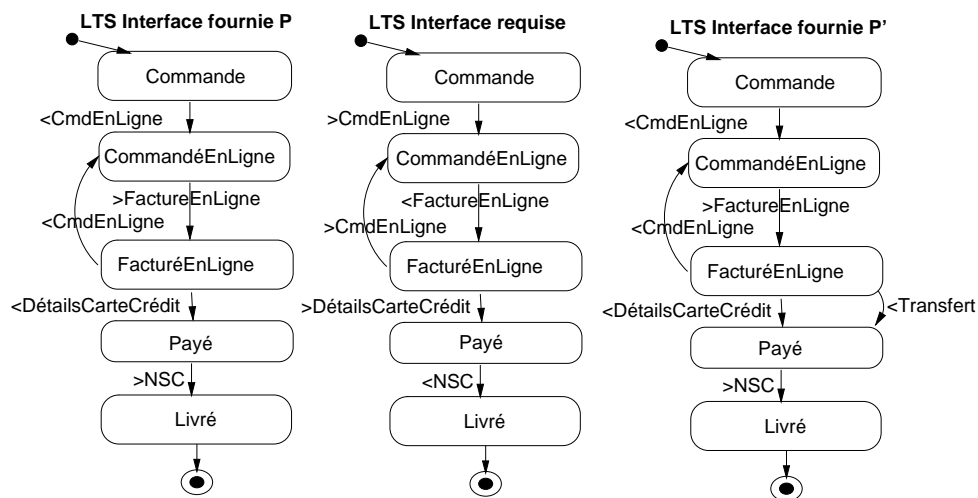


FIG. 3.4 – Simulation des interfaces et conformité

L'interface requise du client est conforme à l'interface fournie P . A chaque envoi de message dans P lui correspond une réception de message dans l'interface requise et *vice-versa*. Etant donné que P' simule P alors l'interface requise du client est également conforme avec P' . En effet, à chaque envoi de message dans P' lui correspond une réception de message dans l'interface requise et *vice-versa*.

3.3.3 Principe de la résolution par médiation

La résolution des incompatibilités s'appuie sur des techniques de médiation. L'idée principale de la médiation est que tout envoi ou réception de messages entre les deux interlocuteurs doit passer par un tiers qui joue le rôle de médiateur [5]. En d'autres termes, si un client envoie un message vers un fournisseur, le message est reçu dans un premier temps par le médiateur avant que ce dernier ne décide de le transmettre à son destinataire, après traitement si nécessaire. Comme illustré dans l'exemple de la figure 3.5, le service fournisseur envoie le message A à destination du client. Néanmoins, ce message est reçu par le médiateur qui le transforme en message B avant de l'envoyer au service client. Le client envoie le message C à destination du service fournisseur. Le message C est intercepté par le médiateur qui le transforme en message D et le transmet au service fournisseur.

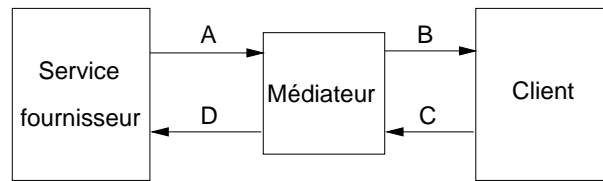


FIG. 3.5 – Principe de la médiation de la conversation

Le rôle de ce médiateur est de garantir la cohérence des interactions entre le client et le fournisseur, c'est-à-dire de réconcilier la conversation initiée par le client selon l'interface qu'il requiert avec celle que le fournisseur fournit. Le médiateur que nous proposons résout les incompatibilités entre la nouvelle version de l'interface fournie P' et les interfaces requises des clients. Les clients continuent à interagir avec le fournisseur en considérant une de ses versions antérieures.

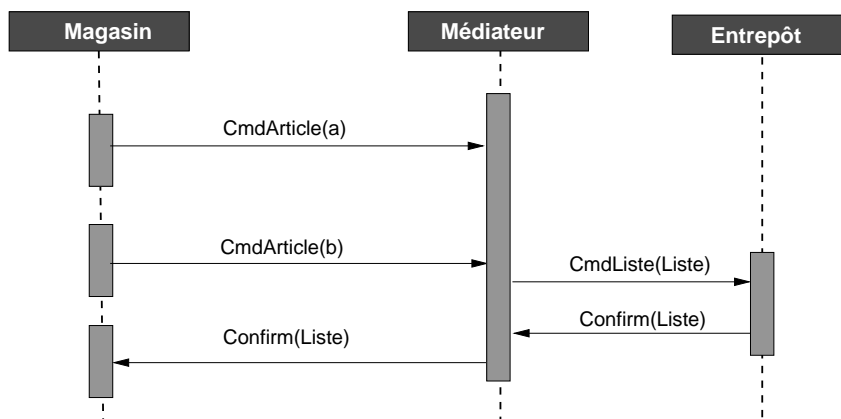


FIG. 3.6 – Diagramme de séquences d'une conversation par médiation

La figure 3.6, illustre un diagramme de séquences d'une conversation entre un fournisseur **Entrepôt** et un client **Magasin** menée par l'intermédiaire d'un médiateur. Le client **Magasin** envoie deux messages consécutifs de commande d'un article (voir dans la figure 3.6, le message `CmdArticle(a)`) puis d'un autre article (voir dans la figure 3.6, le message `CmdArticle(b)`) à destination du fournisseur **Entrepôt**. Ces deux messages sont dans un premier temps interceptés par le médiateur. Ce dernier transforme les messages reçus en une liste des articles commandés puis envoie le message de commande d'une liste d'articles à destination du fournisseur **Entrepôt** (voir dans la figure 3.6, le message `Confirm(Liste)`). Le fournisseur **Entrepôt** envoie le message de confirmation de la commande de la liste d'articles à son client **Magasin**. Le message est reçu par le médiateur qui le fait suivre à son destinataire final, le client **Magasin**. Dans cet exemple, le médiateur a effectué des transformations dans les messages envoyés par le client et les messages reçus par le fournisseur afin de résoudre les incompatibilités entre les interfaces fournies et requises du fournisseur et du client.

3.4 Conclusion

Nous venons dans ce chapitre de présenter les principes de la solution que nous avons développée pour la détection et la résolution d'incompatibilités entre des interfaces de services. Nous avons donné des illustrations de quelques uns des problèmes que nous traitons et nous avons montré comment nous les résolvons. Il est important ici de souligner que le processus de détection que nous proposons identifie toutes les incompatibilités entre deux interfaces, s'il en existe. D'autre part, la solution que nous proposons pour la détection est partielle, elle ne couvre pas tous les cas. Lorsqu'elle peut être mise en œuvre, cette dernière est transparente pour les clients. Les chapitres suivants sont destinés à fournir une formalisation des processus de détection et de résolution.

Chapitre 4

Détection des incompatibilités élémentaires

Sommaire

4.1 Introduction	81
4.2 Incompatibilités élémentaires	82
4.2.1 Notations pour l'expression des incompatibilités	82
4.2.2 Suppression d'une opération	83
4.2.3 Ajout d'une opération	86
4.2.4 Modification d'une opération	88
4.3 Exclusion mutuelle et complétude des expressions	89
4.3.1 Exclusion mutuelle des expressions de détection	90
4.3.2 Etude de la complétude	92
4.4 Algorithme de détection des incompatibilités	93
4.4.1 Principe de l'algorithme de détection	93
4.4.2 Détail de l'algorithme de détection	95
4.4.3 Etude de la complexité de l'algorithme	97
4.5 Conclusion	99

4.1 Introduction

Nous étudions dans ce chapitre une solution pour détecter les différences entre deux interfaces P et P' qui font que le comportement spécifié par P ne simule pas celui spécifié par P' . La phase de détection des incompatibilités est réalisée par un algorithme qui parcourt en parallèle les LTSs associés aux interfaces comparées.

Pour comparer deux interfaces P et P' , nous distinguons :

- Les ajouts d'opérations : les opérations qui sont dans P' et pas dans P .
- Les modifications d'opérations : les opérations de P modifiées dans P' .
- Les suppressions d'opérations : les opérations qui sont dans P et pas dans P' .

Chacun de ces trois cas identifie une différence que nous qualifions d'élémentaire. La section 4.2 détaille et formalise notre solution pour la détection de ces différences, alors que dans la section 4.3 nous discutons de sa complétude. La détection est formalisée par le biais d'un algorithme étudié dans la section 4.4.

4.2 Incompatibilités engendrées par des différences élémentaires

Dans cette section, les cas de détection d'incompatibilités élémentaires sont présentés. Les types d'incompatibilités traités sont : la suppression, l'ajout et la modification d'opérations. Afin de caractériser chaque incompatibilité élémentaire, nous proposons des notations sur les automates qui modélisent chaque incompatibilité (voir la section 4.2.1). Pour détecter les incompatibilités qui font qu'une interface dont le LTS est P' , ne simule pas une autre interface dont le LTS est P , les deux LTSs sont parcourus de manière synchrone. En d'autres termes les paires d'états (s, s') où s (respectivement s') est un des états du LTS P (respectivement P') sont visitées de manière synchrone. Il s'agit de déterminer si une paire courante d'états ne contient pas de différences élémentaires qui font que P' ne simule pas P . Le parcours des deux LTSs ne s'arrête pas à la première incompatibilité rencontrée mais continue, afin de détecter les autres incompatibilités. La suppression d'opération est présentée dans la section 4.2.2. L'ajout d'opération dans une séquence est détaillé dans la section 4.2.3. La modification d'opération est présentée dans la section 4.2.4.

4.2.1 Notations pour l'expression des incompatibilités

Les incompatibilités élémentaires entre deux interfaces fournies sont dues à des différences dites élémentaires qui sont l'ajout, la suppression et la modification d'opérations. Nous rappelons que les différences à détecter entre deux interfaces P et P' sont les différences qui font que P' ne simule pas P . Une incompatibilité élémentaire entre deux interfaces P et P' , signifie qu'il existe une différence élémentaire entre les états et les transitions de P et P' qui font que P' ne simule pas P . Nous introduisons des expressions qui modélisent chacune des incompatibilités élémentaires. L'évaluation de ces expressions est faite par l'algorithme de détection qui prend en paramètres deux LTSs (P

et P') et retourne l'ensemble des incompatibilités élémentaires détectées au niveau des couples d'états (s de P et s' de P'). Une expression est évaluée à *vraie* s'il existe une incompatibilité élémentaire qui vérifie les conditions de l'expression. Ceci traduit le fait que l'incompatibilité élémentaire associée à cette expression est détectée au niveau des couples d'états (s, s').

Les deux interfaces fournies sont définies par deux LTSs P et P' qui sont respectivement décrits par (S, T, s_0, F, L) et (S', T', s'_0, F', L') .

Les opérateurs de base de modélisation des incompatibilités que nous proposons sont⁶ (les exemples s'appuient sur le LTS donné dans la figure 3.1) :

- $s \bullet$: ensemble des transitions sortantes de l'état s .
Par exemple, $\text{FacturéEnLigne} \bullet = \{ (\text{FacturéEnLigne}, <\text{DétailsCarteCrédit}, \text{Payé}), (\text{FacturéEnLigne}, <\text{CmdEnLigne}, \text{CommandéEnLigne}) \}$,
- $t \circ$: état cible de la transition t . Par exemple, $(\text{Payé}, >\text{NSC}, \text{Livré}) \circ = \text{modeLivraison}$,
- $\text{Label}(t)$: libellé de la transition t . Par exemple, $\text{Label}((\text{Payé}, >\text{NSC}, \text{Livré})) = >\text{NSC}$.

L'opérateur \circ (respectivement \bullet) est généralisé aux ensembles de transitions (respectivement d'états). Par exemple, si T est un ensemble de transitions :

$T = \bigcup_{i=1}^n \{t_i\}$ alors $T \circ = \bigcup_{i=1}^n \{t_i \circ\}$, où $n = \|T\|$. L'opérateur Label est également généralisé aux ensembles de transitions.

Les opérateurs ensemblistes sont utilisés pour comparer les LTSs. En voici quelques uns :

- $\|s \bullet\|$: cardinalité de l'ensemble des transitions sortantes de s ,
- $s \bullet - s' \bullet$: différence ensembliste entre les transitions sortantes de s et les transitions sortantes de s' ,
- $s \bullet \subseteq s' \bullet$: les transitions sortantes de s sont incluses dans les transitions sortantes de s' .

Les opérateurs logiques (\wedge, \vee , etc.) sont utilisés pour construire des expressions booléennes. D'autres sélecteurs permettent de construire des ensembles de transitions ou d'états qui sont dans un type donné. Par exemple, l'opérateur *Receptions* appliqué à un ensemble de transitions, permet d'obtenir le sous-ensemble des transitions qui sont associées à des réceptions de messages.

4.2.2 Suppression d'une opération

Lors de la détection des incompatibilités, les LTSs P et P' sont parcourus de manière synchrone et pour chaque paire d'états courante une détection des incompatibilités élémentaires est effectuée.

⁶Ces opérateurs sont inspirés des travaux présentés dans [27].

L'exemple de la figure 4.1 illustre un premier cas de détection de suppression d'opération en comparant les transitions sortantes du couple d'états $(S1, S1')$. Cette comparaison est faite à l'aide de l'opérateur de différence ensembliste. L'idée est de retrouver les libellés des transitions sortantes de l'état $S1$ de l'interface P qui manquent ou qui n'apparaissent pas dans les libellés des transitions sortantes de l'état $S1'$ de l'interface P' . En effet, l'ensemble $Label(S1\bullet) - Label(S1'\bullet)$ contient le libellé $>R(m)$ ce qui traduit que l'opération destinée à envoyer le message $R(m)$ a été supprimée dans l'interface P' .

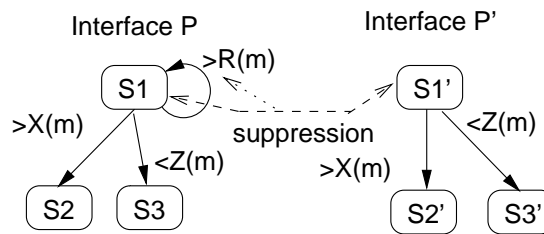


FIG. 4.1 – Suppression d'une opération alternative

Dans cet exemple, d'après le principe du parcours synchrone des LTSs, les prochains couples d'états à considérer dans le processus de détection sont $(S2, S2')$ et $(S3, S3')$. Ce résultat est obtenu en regroupant deux à deux les transitions de $S1\bullet$ et de $S1'\bullet$ dont les libellés sont identiques. Les prochaines paires d'états à étudier sont obtenues par le calcul des états cibles de ces transitions (dont les libellés sont identiques). Un autre couple d'états à considérer dans la détection est $(S1, S1')$. En effet, d'après la transition $(S1, >R(m), S1)$ qui existe dans P mais qui est supprimée dans P' , l'état à considérer est l'état cible de cette transition qui est $S1$. L'état à considérer dans P' est l'état $S1'$. Le résultat de ce parcours est que le couple d'états à visiter est $(S1, S1')$. Cependant, le couple d'états $(S1, S1')$ ne peut être étudié à nouveau car il a déjà été visité. Cette condition assure que le parcours de P et P' se termine et ne boucle pas indéfiniment.

La figure 4.2, illustre un deuxième cas de suppression d'une opération dans une séquence. L'interface fournie P' n'offre pas l'opération $>X(m)$ et enchaîne directement l'exécution de la séquence d'opérations sur l'opération suivante qui est $>Z(m)$.

Dans le processus de détection des incompatibilités, le couple d'états $(S1, S1')$ est caractérisé par une incompatibilité de suppression d'opération. D'une part, le libellé $>X(m)$ de la transition sortante de $S1$ n'apparaît pas dans les libellés des transitions sortantes de $S1'$. D'autre part, le libellé $<Z(m)$ de la transition sortante de l'état $S1'$ apparaît dans les libellés des transitions sortantes de l'état cible de la transition $(S1, >X(m), S2)$. Pour continuer la détection, la progression dans les deux automates se fait uniquement dans l'automate où l'opération a été supprimée. Dans notre exemple, il s'agit de prendre en considération la paire d'états $(S2, S1')$.

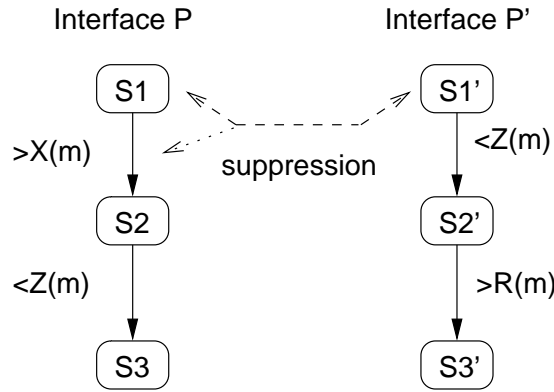


FIG. 4.2 – Suppression d’une opération dans une séquence

La détection de la suppression d’une opération en comparant les transitions sortantes des états courants s et s' respectivement de P et P' est formalisée par les expressions booléennes :

$$\|Label(s\bullet) - Label(s'\bullet)\| \geq 1 \wedge \|Label(s'\bullet) - Label(s\bullet)\| = 0 \quad (4.1)$$

$$\forall \exists t \in s\bullet, \exists t' \in s'\bullet : Label(t) \notin Label(s'\bullet) \wedge ExtIn(t', (t\circ)\bullet) \quad (4.2)$$

Une suppression d’une opération est détectée au niveau d’une paire d’états (s, s') dans deux cas. La ligne 4.1 de l’expression ci-avant exprime le premier cas de la suppression d’une opération. La suppression est détectée lorsque tous les libellés des transitions sortantes de s' existent aussi dans les libellés des transitions sortantes de s (exprimé par $\|Label(s'\bullet) - Label(s\bullet)\| = 0$). De plus, il existe au moins un libellé d’une transition sortante de s qui n’apparaît pas dans les libellés des transitions sortantes de s' (exprimé par $\|Label(s\bullet) - Label(s'\bullet)\| \geq 1$).

La ligne 4.2 de l’expression ci-avant exprime le deuxième cas de la suppression d’une opération. La suppression d’une opération est détectée lorsqu’il existe une paire de transitions sortantes t et t' de leurs états respectifs s et s' telle que : (i) la transition t ne peut être appareillée avec aucune des transitions sortantes de s' (condition exprimée par $\exists t \in s\bullet : Label(t) \notin Label(s'\bullet)$), et (ii) le libellé de t' existe dans le sous-automate de P initialisé à l’état cible de la transition t (exprimé par $\exists t' \in s'\bullet : ExtIn(t', (t\circ)\bullet)$).

Afin de vérifier l’existence d’un libellé d’une transition dans un sous-automate initialisé à l’état cible d’une transition donnée, la fonction d’inclusion étendue est introduite comme suit : $ExtIn(t, T) \equiv T \neq \emptyset \wedge (Label(t) \in Label(T) \vee \bigsqcup_{i=1}^{\|T\|} ExtIn(t, (T_i\circ)\bullet))$. La fonction $ExtIn(t, T)$ (où t est une transition et T est un ensemble de transitions) retourne une valeur *vrai* si : (i) le libellé de la transition t apparaît dans les libellés des transitions de T , ou (ii) il existe une transition t_i de l’ensemble T telle que l’ensemble des transitions sortantes de l’état cible de t_i (soit $T1 = (t_i\circ)\bullet$) et $ExtIn(t, T1)$ retourne la valeur *vrai*.

Afin que cette fonction converge dans le cas d'automates cycliques, une variable des transitions déjà visitées est maintenue.

4.2.3 Ajout d'une opération

Une différence élémentaire entre deux interfaces peut induire une incompatibilité élémentaire dans le cas d'un ajout d'une opération dans une séquence. Cette nouvelle opération est ajoutée dans une interface afin de solliciter des informations complémentaires (propriétaires du client) ou bien pour que le client choisisse de nouvelles options offertes par le fournisseur.

La figure 4.3 illustre un premier cas de détection de l'ajout d'opération au niveau de la paire d'états $(S1, S1')$ où $S1$ (respectivement $S1'$) est un des états du LTS de l'interface P (respectivement de l'interface P'). L'idée est de trouver les libellés des transitions sortantes de l'état $S1'$ qui n'ont pas d'équivalents parmi les libellés des transitions sortantes de l'état $S1$. Dans cet exemple, le résultat de la différence $Label(S1' \bullet) - Label(S1 \bullet)$ est un ensemble qui contient le libellé $>Z(m)$. Ceci signifie qu'une opération d'envoi du message $Z(m)$ a été ajoutée à l'interface P' . En d'autres termes, tous les libellés des transitions sortantes de $S1$ dans l'interface P apparaissent dans les libellés des transitions sortantes de $S1'$ de l'interface P' . Cependant, il existe une transition de libellé $>Z(m)$ qui est ajoutée aux transitions sortantes de $S1'$ et qui n'apparaît pas parmi les libellés des transitions sortantes de $S1$. Cette opération d'envoi du message $Z(m)$ est une opération alternative à l'opération d'envoi du message $X(m)$.

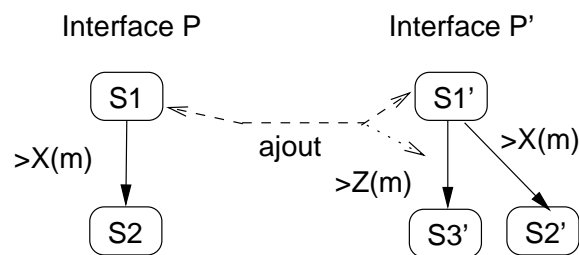


FIG. 4.3 – Ajout d'une opération dans une alternative

L'opération ajoutée est un envoi de message et lorsque le service envoie le message $Z(m)$, l'application cliente doit recevoir ce message sinon l'incompatibilité survient. Cependant, si l'opération ajoutée est une réception de message dans une alternative (soit par exemple, l'ajout de l'opération $>Z(m)$), alors cet ajout n'induit pas d'incompatibilité car c'est une réception optionnelle de message et les applications clientes sont libres de ne pas envoyer ce message.

Un fois que le processus de détection appliqué aux états $S1$ et $S1'$ est terminé, les

paires d'états à étudier ensuite sont $(S2, S2')$ et $(S1, S3')$. Dans la paire d'états $(S1, S3')$, le processus de détection n'avance que dans P' où l'opération a été ajoutée.

Dans l'exemple de la figure 4.4, un deuxième cas d'ajout d'opération peut être détecté au niveau de la paire d'états $(S1, S1')$. En effet la transition sortante $>X(m)$ de $S1$ n'apparaît plus parmi les transitions sortantes de l'état $S1'$. Cependant, l'opération $>X(m)$ apparaît dans la transition sortante de l'état $S2'$. L'état $S2'$ est l'état cible de la transition $(S1', <Y(m), S2')$ dont le libellé $<Y(m)$ est l'opération ajoutée. Autrement dit, l'opération $<Y(m)$ a été ajoutée et précède l'opération $>X(m)$.

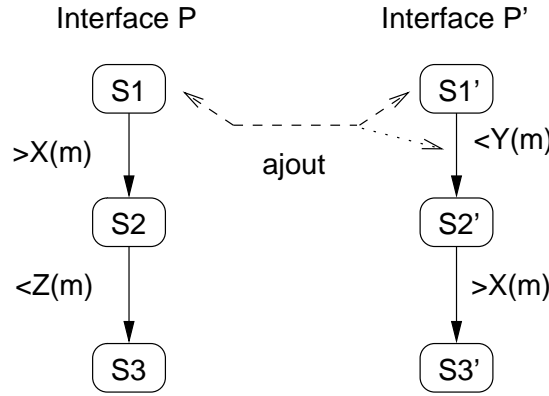


FIG. 4.4 – Ajout d'une opération dans une séquence

Pour continuer la détection, la paire d'états à considérer ensuite est $(S1, S2')$. En effet, le mécanisme de détection ne progresse que dans l'automate P' où l'opération a été ajoutée.

L'ajout d'une opération peut être détecté au niveau d'une paire d'états (s, s') si la condition suivante est vérifiée :

$$(\|Label(s\bullet) - Label(s'\bullet)\| = 0 \wedge \|Label(s'\bullet) - Label(s\bullet)\| \geq 1) \quad (4.3)$$

$$\vee \exists t \in s\bullet, \exists t' \in s'\bullet : Label(t') \notin Label(s\bullet) \wedge ExtIn(t, (t' \circ)\bullet) \quad (4.4)$$

L'ajout d'opération est détecté dans deux cas différents. La ligne 4.3 de l'expression ci-avant représente le premier cas de l'ajout d'une opération dans une alternative. L'ajout d'une opération dans une alternative est détecté lorsqu'il existe une transition sortante de s' dont le libellé n'apparaît pas parmi les libellés des transitions sortantes de s (exprimé par $\|Label(s'\bullet) - Label(s\bullet)\| \geq 1$), tandis que chaque transition sortante de s peut s'appareiller (même libellé) à une transition sortante de s' (exprimé par $\|Label(s\bullet) - Label(s'\bullet)\| = 0$). Dans ce cas, la polarité (envoi ou réception) de l'opération ajoutée doit être vérifiée. En effet, dans le cas de réception de message, l'ajout n'est pas considéré comme induisant une incompatibilité. De ce fait, il ne sera pas retourné en résultat de la détection.

La ligne 4.4 de l'expression ci-avant représente le deuxième cas de l'ajout d'une opération dans une séquence. L'ajout d'une opération dans une séquence est détecté lorsqu'il existe une paire de transitions sortantes t et t' (des états s et s' respectivement) telle que le libellé de t' n'apparaît pas parmi les libellés des transitions sortantes de s (exprimé par $\exists t' \in s' \bullet : Label(t') \notin Label(s \bullet)$). Cependant, le libellé de t apparaît dans le sous-automate de P' initialisé à l'état cible de la transition t' (exprimé par $\exists t \in s \bullet : ExtIn(t, (t' \circ) \bullet)$). Cette condition exprime que l'opération qui est le libellé de la transition t' est ajoutée avant l'opération qui est définie par le libellé de la transition t .

4.2.4 Modification d'une opération

La modification d'une opération X par une autre opération Y signifie que la signature de l'opération X (le nom, les paramètres et le type des paramètres) a été remplacée par la signature d'une autre opération Y . De ce fait, afin de détecter une modification d'opération, il faut comparer les libellés des transitions sortantes des états qui ne sont pas en communs. Ainsi, les transitions dont les libellés sont différents, sont appaireées afin d'être étudiées.

L'exemple de la figure 4.5, illustre l'automate d'une interface fournie P et l'automate d'une autre interface fournie P' . Lors de la détection des incompatibilités qui font que P' ne simule pas P , la paire d'états $(S1, S1')$ possède deux transitions sortantes dont les libellés sont différents ($>X(m)$ et $>Y(m)$). Dans ce cas, l'opération $>X(m)$ n'apparaît plus dans les transitions descendantes de $>Y(m)$ de l'interface P' (ce n'est pas un ajout de l'opération $>Y(m)$). De plus, l'opération $>Y(m)$ n'apparaît pas dans les transitions descendantes de l'opération $>X(m)$ de l'interface P (ce n'est pas une suppression de l'opération $>X(m)$).

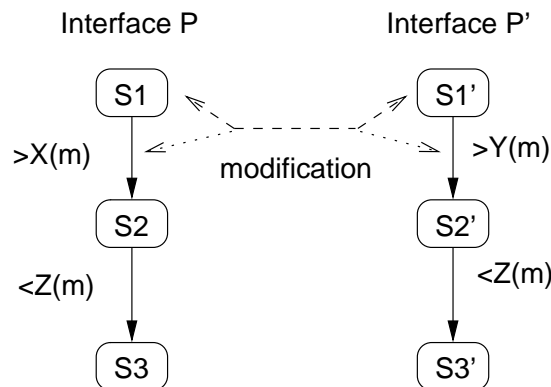


FIG. 4.5 – Cas de détection de la modification d'opération

Pour continuer la détection des incompatibilités, les paires d'états à étudier ensuite sont construites en fonction du type d'incompatibilité détecté. Dans ce cas, il s'agit d'une modification d'opération, le calcul d'une nouvelle paire d'états à étudier est déduit des états cibles des transitions modifiées dans les deux LTS (P et P') simultanément. Dans l'exemple de la figure 4.5, le résultat de ce calcul retourne la paire d'états $(S2, S2')$ à partir de laquelle sera appliquée la détection.

L'expression qui modélise une modification d'opération au niveau d'une paire d'états (s, s') est donnée comme suit :

$$\exists t1 \in s\bullet, \exists t1' \in s'\bullet : Label(t1) \notin Label(s'\bullet) \wedge Label(t1') \notin Label(s\bullet) \quad (4.5)$$

$$\wedge \neg \exists t2 \in s\bullet : ExtIn(t1', (t2\circ)\bullet) \wedge \neg \exists t2' \in s'\bullet : ExtIn(t1, (t2'\circ)\bullet) \quad (4.6)$$

La ligne 4.5 de l'expression ci-avant vérifie qu'il existe une transition $t1$ parmi les transitions sortantes de s dont le libellé n'apparaît pas parmi les libellés des transitions sortantes de s' (exprimé par $\exists t1 \in s\bullet : Label(t1) \notin Label(s'\bullet)$).

Il s'agit également de vérifier qu'il existe une transition sortante $t1'$ de l'état s' dont le libellé n'apparaît pas parmi les libellés des transitions sortantes de s (exprimé par $\exists t1' \in s'\bullet : Label(t1') \notin Label(s\bullet)$).

La ligne 4.6 de l'expression ci-avant vérifie que les autres cas d'incompatibilités d'ajout et de suppression n'apparaissent pas dans ce cas de modification. Il s'agit de vérifier qu'il n'existe pas de transition sortante $t2$ de l'état s telle que le sous-automate de P initialisé à l'état cible de la transition $t2$ ne contient pas le libellé de la transition $t1'$ (exprimé par $\neg \exists t2 \in s\bullet : ExtIn(t1', (t2\circ)\bullet)$). Cette condition exclut le fait que l'opération désignée par le libellé de la transition $t1$ est une opération supprimée de P' . Il s'agit également de vérifier qu'il n'existe pas de transition sortante $t2'$ de l'état s' telle que le sous-automate de P' initialisé à l'état cible de la transition $t2'$ ne contient pas le libellé de la transition $t1$ (exprimé par $\neg \exists t2' \in s'\bullet : ExtIn(t1, (t2'\circ)\bullet)$). Cette condition exclut le fait que l'opération désignée par le libellé de la transition $t1'$ est une opération ajoutée à P' .

4.3 Exclusion mutuelle et complétude des expressions

Les expressions qui modélisent les trois cas d'incompatibilités élémentaires d'une interface s'excluent mutuellement comme démontré dans la section 4.3.1. Chacune des expressions qui modélisent les incompatibilités élémentaires couvrent toutes les incompatibilités élémentaires. Ce résultat est démontré dans la section 4.3.2.

4.3.1 Exclusion mutuelle des expressions de détection

Lorsqu'une différence est détectée au niveau d'une paire d'états (s, s') et qu'elle caractérise des transitions sortantes t et t' de ces deux états, alors cette différence est unique. En d'autres termes, il est impossible d'associer deux différences distinctes à ces états et à ces transitions.

Les trois expressions de détection des changements, énoncés dans les sections précédentes, respectent cette condition. Afin de démontrer ce résultat, l'exclusion mutuelle des expressions de détection doit être vérifiée. En d'autres termes, il faut démontrer que lorsqu'une expression caractérise une paire d'états et leurs transitions sortantes, les deux autres expressions ne sont pas vérifiées pour cette même paire d'états et leurs transitions sortantes. De ce fait, l'exclusion mutuelle est étudiée entre chaque expression associée à une incompatibilité et les deux autres expressions qui sont associées aux autres cas d'incompatibilités.

Soient : Sup , Aj et Mod les expressions de détection de suppression, d'ajout et de modification d'opérations, respectivement. Pour démontrer que Sup , Aj et Mod s'excluent mutuellement il faut démontrer que : $(Sup \oplus Aj) \wedge (Sup \oplus Mod) \wedge (Aj \oplus Mod)$. Une expression A s'exclue mutuellement avec une expression B (noté $A \oplus B$) si et seulement si :

- A est vrai, alors B est faux
- B est vrai, alors A est faux

En d'autres termes A et B ne peuvent pas être vraie simultanément.

Démonstration de $Sup \oplus Aj$:

D'après les expressions de détection d'ajout et de suppression explicitées dans la section 4.2.2 et dans la section 4.2.3, nous avons :

- $Sup = Eq4.1 \vee Eq4.2$,
- $Aj = Eq4.3 \vee Eq4.4$.

$Eq4.1$ (respectivement $Eq4.2$) désigne l'expression de la ligne 4.1 (respectivement 4.2) qui modélise la suppression d'une opération comme présenté dans la section 4.2.2. $Eq4.3$ (respectivement $Eq4.4$) désigne l'expression de la ligne 4.3 (respectivement 4.4) qui modélise l'ajout d'une opération comme présenté dans la section 4.2.3.

De ce fait, pour démontrer que $Sup \oplus Aj$ il faut démontrer que : $(Eq4.1 \oplus Eq4.3) \wedge (Eq4.1 \oplus Eq4.4) \wedge (Eq4.2 \oplus Eq4.3) \wedge (Eq4.2 \oplus Eq4.4)$. Les preuves qui suivent s'appuient sur un raisonnement par l'absurde.

1) Preuve de $Eq4.1 \oplus Eq4.3$:

Supposons que $Eq4.1$ et $Eq4.3$ sont évaluées à vraie pour une même paire d'états s, s' . D'après $Eq4.1$, nous avons $\|Label(s\bullet) - Label(s'\bullet)\| \geq 1$. Or, d'après $Eq4.3$ on a $\|Label(s\bullet) - Label(s'\bullet)\| = 0$. Ainsi, l'hypothèse de départ est fausse car $Eq4.1$ et $Eq4.3$ ne peuvent pas être vraies pour une même paire d'états. On en déduit que $Eq4.1 \oplus Eq4.3$.

2) Preuve de $Eq4.1 \oplus Eq4.4$:

Supposons que $Eq4.1$ est vraie pour une paire d'états (s, s') . Soit t une des transtions sortantes de s dont le libellé $Label(t)$ est celui d'une opération supprimée (le libellé de cette opération n'apparaît pas parmi les libellés des transitions sortantes de s'). Supposons que $Eq4.4$ est vraie pour la même paire d'états (s, s') . Soit t' une des transitions sortantes de s' dont le libellé $Label(t')$ est celui d'une opération ajoutée (le libellé de cette opération n'apparaît pas parmi les libellés des transitions sortantes de s). Supposons $Label(t) = Label(t')$. D'après l'expression $Eq4.1$, $Label(t) \in (Label(s\bullet) - Label(s'\bullet))$. D'après la définition de la différence ensembliste nous avons $Label(t) \in Label(s\bullet) \wedge Label(t) \notin Label(s'\bullet)$. Or, d'après $Eq4.4$, $Label(t') \in Label(s'\bullet)$. Etant donné que $Label(t) = Label(t')$, alors $Label(t) \in Label(s'\bullet)$ (contradiction avec $Label(t) \notin Label(s'\bullet)$). Ainsi, $Eq4.1 \oplus Eq4.4$.

3) Preuve de $Eq4.2 \oplus Eq4.3$:

Supposons que $Eq4.2$ est vraie pour une paire d'états (s, s') . Soit t une transtion sortante de s dont le libellé $Label(t)$ est celui d'une opération supprimée. Supposons que $Eq4.3$ est vraie pour la même paire d'états (s, s') . Soit t' une des transtions sortantes de s' dont le libellé $Label(t')$ est celui d'une opération ajoutée. Supposons également que les libellés des transitions soient identiques ($Label(t) = Label(t')$). D'après l'expression $Eq4.3$, $Label(t') \in Label(s'\bullet) - Label(s\bullet)$. D'après la définition de la différence ensembliste nous avons $Label(t') \in Label(s'\bullet) \wedge Label(t') \notin Label(s\bullet)$. Or, d'après $Eq4.2$, $Label(t) \in Label(s\bullet)$ et étant donné que $Label(t) = Label(t')$, alors $Label(t') \in Label(s\bullet)$ (contradiction avec $Label(t') \notin Label(s\bullet)$). Ainsi, $Eq4.2 \oplus Eq4.3$.

4) Preuve de $Eq4.2 \oplus Eq4.4$:

Supposons que $Eq4.2$ est vraie pour une paire d'états (s, s') . Soit t une transtion sortante de s dont le libellé $Label(t)$ est celui d'une opération supprimée. Supposons que $Eq4.4$ est vraie pour la même paire d'états (s, s') . Soit t' une des transtions sor-

tantes de s' dont le libellé $Label(t')$ est celui d'une opération ajoutée. Supposons que $Label(t) = Label(t')$. D'après Eq4.2 nous avons $Label(t) \in Label(s\bullet)$. Selon Eq4.4 nous avons aussi le fait que $Label(t') \notin Label(s\bullet)$. Or, étant donné que $Label(t) = Label(t')$, nous avons $Label(t') \in Label(s\bullet)$ (contradiction avec $Label(t') \notin Label(s\bullet)$). Ainsi, le résultat Eq4.2 \oplus Eq4.4 est démontré.

Conclusion de la démonstration de $Sup \oplus Aj$:

D'après les preuves (1), (2), (3) et (4) nous avons $(Eq4.1 \oplus Eq4.3) \wedge (Eq4.1 \oplus Eq4.4) \wedge (Eq4.2 \oplus Eq4.3) \wedge (Eq4.2 \oplus Eq4.4) \Rightarrow Sup \oplus Aj$. Les expressions de détection d'ajout et de suppression d'opérations s'excluent mutuellement.

La démonstration de $Sup \oplus Mod$ et de $Aj \oplus Mod$ est analogue à la démonstration de $Sup \oplus Aj$ (voir l'annexe A).

4.3.2 Etude de la complétude

Les expressions de détection des incompatibilités élémentaires d'ajout, de suppression et de modification d'opérations couvrent tous les cas d'incompatibilités élémentaires qui existent. En d'autres termes, l'expression de détection d'un ajout (respectivement d'une suppression et d'une modification) d'une opération modélisent toutes les incompatibilités dues à un ajout (respectivement suppression et modification) d'une opération.

L'expression qui modélise un ajout d'une opération permet de détecter tous les ajouts d'opérations qui existent en comparant les ensembles des transitions sortantes d'une paire d'états. D'après les résultats de la théorie des ensembles, si $x \notin S$ alors $x \in S \cup \{x\}$, où S est un ensemble d'éléments et x est un élément quelconque. Posons $S' = S \cup \{x\}$, où S' est un autre ensemble d'éléments. On dit alors que x est ajouté à l'ensemble S pour obtenir S' . Pour trouver les éléments qui ont été ajoutés à l'ensemble S afin d'obtenir l'ensemble S' , la différence ensembliste $S' - S$ est calculée. Le résultat obtenu est l'élément x . Ce même raisonnement a été utilisé dans la formalisation de l'expression d'un ajout d'une opération.

L'expression qui modélise la suppression d'une opération permet de détecter toutes les suppressions d'opérations qui existent en comparant les ensembles des transitions sortantes d'une paire d'états. D'après les résultats de la théorie des ensembles, si $x \notin S'$ alors $x \in S' \cup \{x\}$, où S' est un ensemble d'éléments et x est un élément quelconque. Posons $S = S' \cup \{x\}$, où S est un autre ensemble d'éléments. D'après l'opérateur de la différence ensembliste, nous avons $S' = S - \{x\}$.

On dit alors que x est supprimé de l'ensemble S pour obtenir S' . Ce même raisonnement a été utilisé dans la formalisation de l'expression de la suppression d'une opération.

L'expression qui modélise la modification d'une opération permet de détecter toutes les modifications d'opérations qui existent en comparant les ensembles des transitions sortantes d'une paire d'états. D'après les résultats de la théorie des ensembles, si $x \notin S$ alors $x \in S \cup \{x\}$ et si $y \notin S$ alors $y \in S \cup \{y\}$. S est un ensemble d'éléments, et x et y sont des éléments quelconques. Posons $S1 = S \cup \{x\}$ et $S2 = S \cup \{y\}$, où $S1$ et $S2$ sont deux autres ensembles d'éléments. Si $x \neq y$, alors d'après l'opérateur de la différence ensembliste, nous avons $S1 - S2 = \{x\}$ et $S2 - S1 = \{y\}$. On dit alors que l'élément x de l'ensemble $S1$ est modifié par l'élément y pour obtenir l'ensemble $S2$. Ce même raisonnement a été utilisé dans la formalisation de l'expression de la modification d'une opération.

4.4 Algorithme de détection des incompatibilités élémentaires

Le principe de cet algorithme est de parcourir de manière synchrone les deux automates des deux interfaces P et P' puis de les comparer afin de trouver les différences qui font que P' ne simule pas P . Ce principe est illustré dans la section 4.4.1. Le détail de l'algorithme est donné dans la section 4.4.2. Une étude sur le complexité de l'algorithme est présentée dans la section 4.4.3.

4.4.1 Principe de l'algorithme de détection

L'algorithme qui réalise la détection des incompatibilités induites par des différences élémentaires s'appuie sur la comparaison de deux LTSs qui modélisent les deux interfaces fournies. Soient P_j le LTS qui modélise l'interface d'un service et P_i le LTS qui modélise l'interface d'un autre service. Le principe de l'algorithme est de parcourir de manière synchrone les deux LTSs P_i et P_j puis de retrouver tous les différences élémentaires (ajout, suppression et modification d'opération) qui font que P_j ne simule pas P_i .

L'algorithme vise à retrouver tous les couples d'états s_i et s_j (de P_i et P_j respectivement) qui satisfont les expressions de détection d'incompatibilités induites par les différences élémentaires. Si un ajout est détecté, l'algorithme détermine la paire d'états à visiter en ne considérant que l'état cible de la transition de l'opération ajoutée dans l'interface P_j , tandis que l'état courant de l'interface P_i reste inchangé dans cette nou-

velle paire d'états à examiner. L'exemple de la figure 4.6 illustre la détection de l'ajout de l'opération $\langle Y(m) \rangle$ au niveau de la paire d'états $(S1, S1')$. L'algorithme ne progresse que dans P_j alors la prochaine paire d'états à examiner est $(S1, S2')$. En examinant $(S1, S2')$, aucune incompatibilité n'est détectée. La prochaine paire d'état à étudier $(S2, S3')$ est obtenue en progressant dans P_i et P_j simultanément. Lorsqu'il s'agit d'une *suppression* d'opération, contrairement à l'ajout, la nouvelle paire d'états à examiner est déduite à partir de l'état cible de la transition de l'opération supprimée dans l'interface P_i et de l'état courant de l'interface P_j . L'exemple de la figure 4.6 illustre la détection de l'ajout de l'opération $\langle R(m) \rangle$ au niveau de la paire d'états $(S2, S3')$. L'algorithme ne progresse que dans P_i alors la prochaine paire d'états à examiner est $(S4, S3')$. Lorsque les libellés des transitions sont identiques, aucune incompatibilité ne leur est associée. L'algorithme progresse simultanément dans les deux automates P_i et P_j . Dans ce cas, la prochaine paire d'états à visiter est $(S3, S4')$. Si une modification d'opération est détectée, alors la nouvelle paire d'états à étudier est déduite des états cibles des transitions des opérations modifiées aussi bien dans P_i que dans P_j . L'exemple de la figure 4.6 illustre la détection de la modification de l'opération $\langle K(m) \rangle$ dans P_i par une autre opération $\langle W(m) \rangle$ dans P_j au niveau de la paire d'états $(S3, S4')$. L'algorithme progresse simultanément dans les deux automates P_i et P_j . Dans ce cas, la prochaine paire d'états à visiter est $(S5, S5')$ (états puits).

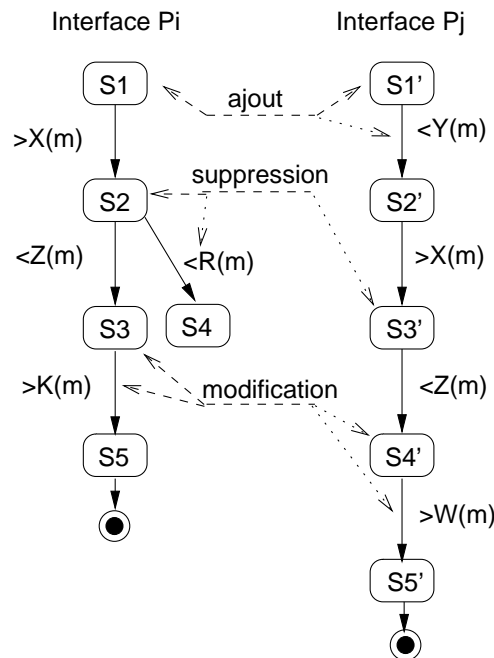


FIG. 4.6 – Principe de progression de l'algorithme

4.4.2 Détail de l'algorithme de détection

L'algorithme, donné dans la figure 4.7 met en œuvre une recherche en profondeur des deux LTSs comparés, où sont utilisées des variables de type pile de paires d'états visitées (pour les paires d'états déjà examinées) et àVisiter (pour les paires d'états des deux LTSs qu'il reste à examiner) (voir ligne 5). Le type PaireEtats est défini de telle sorte que le premier état appartient au LTS de l'interface P_i et le second état du couple appartient au LTS de l'interface P_j . La définition de ces types est donnée ci-après :

PaireEtats : le type $\langle \text{Etat}, \text{Etat} \rangle$
 Pile de PaireEtats : le type $\text{Pile}\langle \text{PaireEtats} \rangle$

L'algorithme de détection retourne un ensemble de différences entre P_i et P_j qui font que P_j ne simule pas P_i . La définition du type qui modélise une différence est donné ci-après :

Change : le type $\langle \text{Etat}, \text{Transition}, \text{Etat}, \text{Transition} \rangle$

Le résultat de l'algorithme de détection est un ensemble de tuples $\langle si, ti, sj, tj \rangle$ du type Change où ti et tj sont des transitions sortantes de si et sj qui peuvent être sans valeur selon le changement détecté. Si un ajout est détecté au niveau de la paire d'états (si, sj) , alors ti prend la valeur nulle et tj est la transition dont le libellé désigne l'opération ajoutée. Dans le cas de détection de suppression, tj est sans valeur et ti est la transition dont le libellé désigne l'opération supprimée. Si une modification est détectée, alors ti est la transition dont le libellé représente l'opération qui a été modifiée par l'opération désignée par le libellé de la transition tj .

La première paire d'états à visiter est celle qui contient l'état initial de P_i et l'état initial de P_j (voir la ligne 6). Une paire d'états n'est visitée qu'une seule fois. Pour garantir cette condition, l'algorithme utilise la variable visitées qui sauvegarde les paires d'états déjà visitées (voir la ligne 10).

Les libellés en commun des transitions sortantes de si et des transitions sortantes de sj sont considérés comme n'induisant pas d'incompatibilité, et la progression dans P_i et P_j pour déterminer les prochaines paires d'états à visiter se fait de manière simultanée. Ainsi, l'algorithme construit un ensemble combEgal de paires d'états où les états sont des états cibles des transitions équivalentes (voir la ligne 11). Deux transitions sont équivalentes lorsque leurs libellés sont identiques. Les différences entre les libellés des transitions sortantes de si et de sj sont également calculées (voir la ligne 12). Les deux ensembles

```

1  Detection (Pi : LTS ; Pj : LTS ) : {Change}
2  { Detection(Pi,Pj) retourne un ensemble de tuples de changements où chaque tuple est de la
   forme < si, ti, sj, tj >. Tandis que si and sj représentent les états de Pi et Pj respectivement,
   ti et tj sont soit des valeurs nulles soit des transitions sortantes de si et sj respectivement }
3  ensRes : {Change}; { variable résultat }
4  si, sj : Etat; { variables intermédiaires }
5  visitées, àVisiter : Pile de PaireEtats; { paires d'états déjà visitées / à visiter }
6  si ← initEtat(Pi); sj ← initEtat(Pj)
7  àVisiter.empiler((si, sj))
8  Tant que nonVide(àVisiter)
9    (si, sj) ← àVisiter.dépiler();
10   visitées.empiler( (si, sj) ) { ajouter la paire d'états courante à la pile des paires d'états déjà
   visitées }
11   combEgal ← {(ti, tj) ∈ si• × sj• | Label(ti) = Label(tj)} { transitions équivalentes }
12   difPiPj ← {ti ∈ si• | Label(ti) ∉ Label(sj•)};
   difPjPi ← {tj ∈ sj• | Label(tj) ∉ Label(si•)}
13   combPiPj ← difPiPj × difPjPi; { toutes les paires de transitions sortantes de si et sj qui
   ne sont pas équivalentes }
14   Si ||difPiPj|| ≥ 1 et ||difPjPi|| = 0 alors { suppression }
15     Pour chaque t dans difPiPj faire ensRes.ajouter(< si, t, sj, null >)
16     Si((to, sj) ∉ visitées) alors àVisiter.empiler((to, sj))
17   Si ||difPjPi|| ≥ 1 et ||difPiPj|| = 0 alors { ajout }
18     Pour chaque t dans difPjPi faire
19       Si (polarité(t) = 'envoi') alors ensRes.ajouter(< si, null, sj, t >)
   { dans le cas contraire, cet ajout n'induit pas d'incompatibilité }
20       Si ((si, to) ∉ visitées) alors àVisite.empiler((si, to))
21   Pour chaque (ti, tj) dans combPiPj faire
22     Si ExtIn(ti, (tj◦)•) alors { ajout }
23       ensRes.ajouter(< si, null, sj, tj >)
24       Si ((si, tj◦) ∉ visitées) alors àVisiter.empiler((si, tj◦))
25     Si ExtIn(tj, (tio)•) alors { suppression }
26       ensRes.ajouter(< si, ti, sj, null >)
27       Si ((tio, sj) ∉ visitées) alors àVisiter.empiler((tio, sj))
28     Si ( (¬∃tj' ∈ sj• : ExtIn(ti, (tj'◦)•) ) ∧ (¬∃ti' ∈ si• : ExtIn(tj, (ti'◦)•) ) ) alors
29       ensRes.ajouter(< si, ti, sj, tj >) { modif. }
30       Si((tio, tj◦) ∉ visitées) alors àVisiter.empiler((tio, tj◦))
31   Pour chaque (ti, tj) dans combEgal faire
   Si ((tio, tj◦) ∉ visitées) alors àVisiter.empiler((tio, tj◦))
32  Retourner ensRes

```

FIG. 4.7 – Algorithme de détection des incompatibilités élémentaires

des différences de libellés des transitions sont mis dans les variables diffPiPj (transitions dont les libellés apparaissent dans $\text{Label}(si\bullet)$ mais qui n'apparaissent pas dans $\text{Label}(sj\bullet)$) et diffPjPi (transitions dont les libellés apparaissent dans $\text{Label}(sj\bullet)$ mais n'apparaissent pas dans $\text{Label}(si\bullet)$). La ligne 13 illustre la construction des combinaisons des transitions (paires de transitions) dont les libellés ne sont pas en commun à $\text{Label}(si\bullet)$ et $\text{Label}(sj\bullet)$. Ces combinaisons sont indiquées par le variable combPiPj .

Les lignes de 14 à 16 décrivent la détection de la différence par une suppression d'opération lorsqu'une transition sortante de si ne correspond à aucune transition dans $sj\bullet$. Le résultat retourné est un ensemble de tuples $\langle si, t, sj, null \rangle$ où t est l'une des transitions sortantes de si dont le libellé n'apparît pas parmi les libellés des transitions sortantes de sj . Lorsqu'une opération est supprimée, l'algorithme n'avance que dans la branche de la transition supprimée dans le LTS Pi , tandis que le même état dans le LTS Pj est toujours pris en compte dans le calcul des paires d'états à visiter.

La détection d'une incompatibilité par un ajout d'opération est similaire à la détection d'un changement par suppression d'opération (voir les lignes de 17 à 20). Toutefois, pour chaque transition t dont le libellé est l'opération ajoutée, la polarité de t est vérifiée. Si la polarité de l'opération ajoutée est *envoi*, alors la différence par ajout de cette opération induit une incompatibilité (voir la ligne 19). Dans le cas contraire (réception de message), l'ajout de cette opération n'induit pas d'incompatibilité et n'est pas pris en compte dans le résultat global.

La variable combPiPj contient des paires de transitions telle que le libellé de la première transition ti appartient à $\text{Label}(si\bullet)$ mais n'appartient pas à $\text{Label}(sj\bullet)$ tandis que le libellé de la deuxième transition tj appartient à $\text{Label}(sj\bullet)$ mais n'appartient pas à $\text{Label}(si\bullet)$. Ainsi, pour chacun de ces couples de transitions l'algorithme vérifie l'incompatibilité par *ajout* (voir les lignes de 22 à 24), l'incompatibilité par *suppression* (voir les lignes de 25 à 27) et l'incompatibilité par *modification* d'opérations (voir les lignes 28 à 30).

En dernier lieu, les transitions dont les libellés sont identiques ne présentent aucune incompatibilité. Les nouvelles paires d'états à étudier sont déduites de manière symétrique en considérant les états cibles de chaque paire de transitions de l'ensemble combEgal (voir la ligne 31).

4.4.3 Etude de la complexité de l'algorithme

Dans cette section nous étudions la complexité de l'algorithme afin de le comparer le aux algorithmes présentés dans [100, 74].

Soient I_1 et I_2 deux LTSs qui décrivent les interfaces comportementales de deux services. Les valeurs de n_1 et n_2 représentent les nombres des états de I_1 et I_2 respectivement et m_1 et m_2 les nombres des transitions de I_1 et I_2 respectivement. Soient, également, w_1 et w_2 les nombres des transitions de libellés distincts de I_1 et I_2 respectivement. L'algorithme effectue une recherche en profondeur de l'espace composé des paires d'états (s_1, s_2) telle que s_1 est un état de I_1 et que s_2 est un état de I_2 . L'algorithme visite une paire d'états au plus une seule fois. De ce fait, dans le pire des cas où les paires d'états sont issues du produit cartésien des états de I_1 et I_2 , et sont toutes atteignables, une première évaluation de la complexité de l'algorithme fournirait $O(n_1 * n_2)$. Un autre fait à constater est que pour chaque paire d'états, l'algorithme examine des paires de transitions (t_1, t_2) telle que t_1 est une transition sortante de s_1 et t_2 est une transition sortante de s_2 . Une transition t d'un LTS, qui ne peut correspondre à aucune autre transition d'un autre LTS, est examinée à part. Chaque paire de transitions (t_1, t_2) est examinée au plus une seule fois. Ainsi, dans le pire des cas où les paires de transitions sont issues du produit cartésien des transitions de I_1 et I_2 , et sont toutes atteignables, une deuxième évaluation de la complexité de l'algorithme est $O(m_1 * m_2 + m_1 + m_2)$.

Lorsqu'une paire de transitions (t_1, t_2) est examinée, l'algorithme évalue une condition. Cette condition est évaluée sur les libellés des transitions, dans certains cas. L'évaluation d'une condition peut aussi s'appuyer sur une fonction d'observation en aval des transitions. L'objectif de cette fonction est de vérifier si pour un libellé donné dans un automate il existe un libellé identique dans un autre sous-automate initialisé à l'état cible de t_1 ou de t_2 . Cette fonction utilise un algorithme de recherche en largeur dont la complexité est de $O((n_1 + m_1) * w_1)$ pour le LTS I_1 , et est de $O((n_2 + m_2) * w_2)$ pour le LTS I_2 . Etant donné que le nombre de transitions distinctes est majoré par le nombre de transitions alors w_1 (respectivement w_2) est remplacé par m_1 (respectivement m_2) et on obtient une complexité est de $O((n_1 + m_1) * m_1)$ (respectivement $O((n_1 + m_1) * m_2)$) pour le LTS I_1 . Ainsi, la complexité de cette fonction est $O(n_1 * m_1 + (m_1)^2 + n_2 * m_2 + (m_2)^2)$.

En combinant les trois complexités obtenues

$O(n_1 * n_2)$, $O(m_1 * m_2 + m_1 + m_2)$ et $O(n_1 * m_1 + (m_1)^2 + n_2 * m_2 + (m_2)^2)$, la complexité globale est $O(n_1 * m_1 + (m_1)^2 + n_2 * m_2 + (m_2)^2 + n_1 * n_2 + m_1 * m_2)$. En supposant que le nombre de transitions est plus grand que le nombre d'états, la complexité sera majorée par $O((m_1 + m_2)^2)$. Ainsi, la complexité de l'algorithme de détection est quadratique au nombre total des transitions des deux LTSs.

La complexité de notre algorithme est meilleure que celle des algorithmes présentés dans les travaux [100, 74]. La complexité de leurs algorithmes sont exponentielle ou factorielle. Dans le chapitre 6 une étude expérimentale complète cette comparaison théorique.

4.5 Conclusion

Notre démarche de détection des incompatibilités nous a amenés à formaliser la détection des cas d'incompatibilités élémentaires : l'ajout, la suppression et la modification. Dans l'ajout (la suppression) d'une opération, nous avons distingué l'ajout (respectivement la suppression) d'une opération dans une alternative, et de l'ajout (respectivement la suppression) d'une opération dans une séquence. Cependant, l'ajout d'une opération n'induit pas nécessairement d'incompatibilité notamment lorsque l'opération ajoutée est la réception optionnelle d'un message. La modélisation des incompatibilités élémentaires garantit la complétude des expressions associées aux incompatibilités élémentaires. Chaque expression s'exclut mutuellement avec les deux autres expressions. La détection des incompatibilités est réalisée par un algorithme qui parcourt de manière synchrone les deux LTSs et qui évalue les expressions de détection des incompatibilités pour chaque paire d'états visitée. Les résultats retournés par l'algorithme de détection renseignent avec précision sur les localisations des incompatibilités sur les deux LTSs comparés. Le chapitre suivant est consacré à l'étude de la résolution des incompatibilités détectées entre deux interfaces. Nous avons identifié certains cas où la résolution est automatisable. L'objet du chapitre 5 est de détailler l'approche que nous proposons pour cela, et qui s'appuie sur un processus de génération de médiateurs.

Chapitre 5

Résolution des incompatibilités par médiation

Sommaire

5.1	Introduction	101
5.2	Incompatibilités automatiquement résolubles	102
5.2.1	Suppression d'un choix d'opération dans une alternative	102
5.2.2	Suppression de l'acquiescement d'une opération	103
5.3	Résolution des incompatibilités	104
5.3.1	Cas de suppression d'un choix d'opération	105
5.3.2	Cas de suppression d'un acquiescement d'opération	106
5.3.3	Algorithme de résolution des incompatibilités	107
5.4	Génération automatique de médiateurs	109
5.4.1	Interactions via des médiateurs	109
5.4.2	Utilisation de médiateurs	110
5.4.3	Sélection des médiateurs	113
5.5	Conclusion	115

5.1 Introduction

Pour la résolution des incompatibilités nous considérons le contexte de l'interface P d'un service qui évolue vers une nouvelle définition P' de son interface. Lorsque P' ne simule pas P , les interfaces requises des clients du service qui sont conformes avec la définition de l'interface fournie P , ne seront plus conformes avec la nouvelle définition de l'interface fournie P' . Dans ce contexte, la résolution vise à maintenir la compatibilité

des interfaces requises des clients avec chaque nouvelle définition de l'interface fournie du service. Chaque nouvelle définition de l'interface fournie est considérée comme une nouvelle version de cette interface fournie.

Dans l'ensemble des incompatibilités qui font que P' ne simule pas P , il est possible de distinguer certains cas où les incompatibilités sont automatiquement résolubles. Les expressions qui caractérisent ces cas sont présentées dans la section 5.2. La résolution s'appuie sur un processus de génération de médiateurs tel que illustré dans la section 5.3. Plusieurs médiateurs peuvent être générés pour résoudre les incompatibilités entre les clients et le service. Une mesure de similarité des interfaces est introduite dans la perspective de sélectionner le médiateur approprié dans une conversation entre un client et le service (voir la section 5.4)

5.2 Incompatibilités automatiquement résolubles

Pour découvrir les incompatibilités qui peuvent être automatiquement résolues, il faut en étudier le contexte et aussi vérifier qu'il est possible de générer le médiateur qui sera capable de les corriger. Les expressions qui modélisent les incompatibilités automatiquement résolubles s'appuient sur l'ensemble des nuplets des résultats de la détection des incompatibilités élémentaire. Deux cas d'incompatibilités automatiquement résolubles sont présentés dans les sections 5.2.1 et 5.2.2.

5.2.1 Suppression d'un choix d'opération dans une alternative

Lorsqu'une opération apparaît comme un terme supplémentaire à une alternative (opération de réception) dans la nouvelle définition de l'interface fournie d'un service cela ne génère aucune incompatibilité. En effet, le choix qui en découle peut ne pas être considéré dans l'interface requise des clients. La différence sera détectée alors qu'elle n'a pas d'incidence sur les conversations. Néanmoins, si un choix d'opération quelconque dans une alternative est supprimé d'une interface fournie, alors les incompatibilités seront générées lors des interactions avec les clients qui continuent à solliciter cette opération supprimée.

La figure 5.1 illustre la situation où une opération offerte dans un choix d'opérations est supprimée. Dans l'interface P , l'opération $c : Z(m)$ participe à une alternative, avec d'autres opérations, toutes correspondant à une réception de messages. Comme indiqué dans la figure, cette opération n'apparaît plus dans l'interface P' . Ce qui traduit le fait que le terme de l'alternative ne peut pas, dans P' , être choisi par les clients.

Cette incompatibilité est détectée en s'appuyant sur le résultat de la détection de l'incompatibilité de suppression d'une opération dans une séquence. L'incompatibilité de

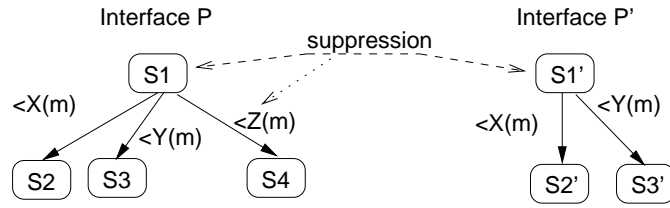


FIG. 5.1 – Suppression d'un terme d'une alternative

suppression d'opération est dans l'ensemble *ensRes* des incompatibilités retourné comme résultat de l'algorithme de détection (voir la section 4.4). Ainsi le cas de suppression d'un terme d'une alternative est modélisé par la formule suivante :

$\exists C \in \text{ensRes} :$

$$C.\text{change} = \text{'suppression'} \wedge \|\text{Réceptions}((C.si)\bullet)\| \geq 2 \wedge \text{Polarité}(C.ti) = \{<\}$$

D'après cette formule, la détection de suppression d'un choix dans une alternative consiste à vérifier, dans le résultat d'une incompatibilité de suppression d'opération, si les transitions sortantes de $C.si$ (état de la définition de l'interface P où la suppression d'opération est détectée) contiennent des réceptions de messages (exprimé par la condition $\|\text{Réceptions}((C.si)\bullet)\| \geq 2$). L'autre condition à vérifier est que l'opération supprimée soit une opération de réception de message (formalisée par $\text{Polarité}(C.ti) = \{<\}$).

5.2.2 Suppression de l'acquittement d'une opération

L'incompatibilité de suppression d'une opération d'acquittement est une suppression d'opération qui désigne un acquittement d'une autre opération. Dans la définition d'une interface fournie, une opération d'acquittement est un envoi de message qui vient après une opération de réception d'un autre message. La sémantique de l'acquittement doit être précisée dans le contenu du message envoyé.

La figure 5.2 illustre la suppression d'une opération d'envoi d'acquittement $>Y(m : \text{acq})$ précédée d'une autre opération de réception du message $X(m)$.

La détection de l'incompatibilité de suppression d'un acquittement est traduite par l'expression booléenne suivante :

$\exists C \in \text{ensRes} :$

$$C.\text{change} = \text{'suppression'} \wedge (\exists t \in \bullet(C.si) : \text{Polarité}(t) = \{<\}) \\ \wedge \text{Polarité}(C.ti) = \{>\} \wedge \text{Acq}(\text{Message}(C.ti))$$

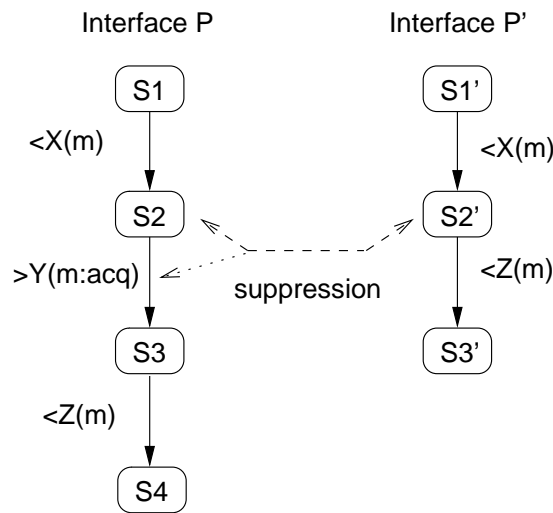


FIG. 5.2 – Suppression d'un acquittement d'une opération

Dans la détection de l'incompatibilité de suppression d'un acquittement d'opération, l'état qui contient la transition sortante de l'opération supprimée doit avoir au moins une transition entrante dont la polarité est une réception de message. Cette condition est exprimée par $\exists t \in \bullet(C.si) : Polarité(t) = \{<\}$. En effet, dans cette incompatibilité, un acquittement vient en réponse à une réception d'un message. Pour vérifier que l'opération supprimée est bien un envoi d'un message d'acquittement, le message de la transition supprimée est testé à l'aide de la méthode $Acq(Message(C.ti))$. Cette dernière retourne la valeur *vrai* si le type du message donné en paramètre est un acquittement, et elle retourne *faux* dans le cas contraire.

5.3 Résolution des incompatibilités

La résolution des incompatibilités qui font qu'une interface P' ne simule pas une autre interface P est effectuée par un médiateur automatiquement généré. Le principe de la résolution est basé sur la création d'un nouveau LTS qui modélise le comportement du médiateur. Le LTS est, dans un premier temps, une copie de l'automate de l'interface P' . Dans un second temps, des modifications de ce LTS sont introduites par des ajouts de transitions et d'états selon l'incompatibilité à résoudre afin que le LTS ainsi obtenu simule le comportement de P . Quelques cas qu'il est possible de résoudre automatiquement sont présentés dans les sections 5.3.1 et 5.3.2. Un algorithme de résolution des incompatibilités est présenté dans la section 5.3.3.

5.3.1 Cas de suppression d'un choix d'opération

Pour résoudre l'incompatibilité de suppression d'un choix d'opération qui fait que le LTS P' d'une interface ne simule pas le comportement d'un LTS P d'une autre interface, le médiateur est défini avec une interface comportementale qui contient ce choix d'opération. Toutefois, cette opération est toujours suivie d'une opération d'envoi d'un message de refus. Ainsi, le client aura la possibilité (s'il l'a prévu dans son interface requise) de choisir une autre opération. Comme illustré dans la figure 5.3, le client qui effectue un paiement en espèces, reçoit un message de refus tant qu'il ne choisit pas un paiement par transfert ou par chèque. En effet, le LTS d'un médiateur prend en compte les réceptions de message pour des paiement en espèce mais les refuse systématiquement.

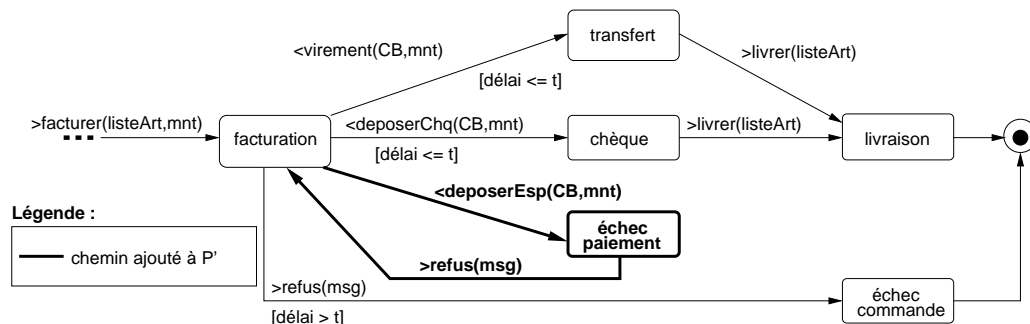


FIG. 5.3 – LTS du médiateur pour la suppression d'un choix d'opération

L'interface du médiateur est construite en ajoutant au LTS P' un chemin de retour à l'état où l'opération est supprimée. Chaque chemin de retour correspond à un choix supprimé et inclut l'envoi d'un message de refus au client.

La méthode qui résout l'incompatibilité de suppression d'un choix d'une opération dans une alternative construit le LTS du médiateur. Ce dernier est construit pour résoudre l'incompatibilité C de suppression d'opération qui fait que le LTS P_j d'une interface ne simule pas le LTS P_i .

La figure 5.4 illustre la méthode de résolution de suppression d'un terme d'une alternative qui fait que P_j ne simule pas P_i . Le résultat de cette méthode est un LTS construit à partir du LTS P_j . Un état intermédiaire est ajouté à ce dernier (voir la ligne 4). Deux transitions, sont ajoutées au LTS P_j . Une première transition porte le libellé de l'opération supprimée (voir la ligne 5). Une deuxième transition a un libellé d'une opération d'envoi d'un message de refus (voir la ligne 6).

- 1 RésoudreAlt (C : Change ; Pi : LTS ; Pj : LTS)
- 2 { RésoudreAlt(P,Pi,Pj) : L'incompatibilité élémentaire P est une suppression d'un terme d'une alternative. Le résultat est la modification du LTS Pj par l'ajout d'un état et d'un ensemble de transitions. }
- 3 S : État { Variable intermédiaire }
- 4 AjouterÉtat(Pj,S) { Ajoute un état S à l'ensemble des états de Pj }
- 5 AjouterTransition(Pj,C.sj,Label(C.ti),S) { Ajoute une transition C.ti au LTS Pj dont l'état source est C.sj et l'état cible est S }
- 6 AjouterTransition(Pj,S,'>refus(msg)',C.sj) { Ajoute une transition d'envoi d'un message de au LTS Pj dont l'état source est S et l'état cible est C.sj }

FIG. 5.4 – Méthode de résolution de suppression d'un terme d'une alternative

5.3.2 Cas de suppression d'un acquittement d'opération

Pour résoudre l'incompatibilité de suppression d'un acquittement d'opération qui fait le LTS P' d'une interface ne simule pas le comportement d'un LTS P d'une autre interface, le médiateur est défini avec une interface comportementale qui contient cette opération d'acquittement. Ainsi, le médiateur doit avoir dans son interface une opération qui offre la possibilité d'envoyer, l'acquittement attendu par les clients.

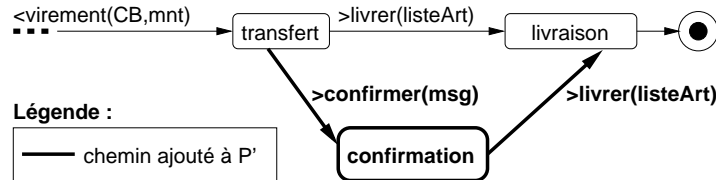


FIG. 5.5 – LTS du médiateur pour la suppression d'un acquittement

Dans l'exemple de la figure 5.5, la transition d'envoi de confirmation suivie de la livraison est ajoutée au niveau de l'état *transfert* où la suppression est détectée. Ce chemin est emprunté par le médiateur dans le cas où l'opération de réception de la confirmation est prévue dans l'interface requise du service client.

La méthode qui résout l'incompatibilité de suppression d'un choix d'une opération dans une alternative construit le LTS du médiateur. Ainsi, l'interface du médiateur est construite en ajoutant un chemin qui contient une transition de l'opération d'acquittement et une transition de l'opération qui suit l'acquittement. La figure 5.6 illustre la méthode de résolution de suppression d'une opération d'acquittement qui fait que Pj ne simule pas Pi . Le résultat de cette méthode est le LTS du médiateur construit à partir du LTS

P_j .

- 1 RésoudreAcq (C : Change; P_i : LTS ; P_j : LTS)
- 2 { RésoudreAcq(P, P_i, P_j) : L'incompatibilité élémentaire C est une suppression d'acquittement. Le résultat est la modification du LTS P_j par ajout d'un état et d'un ensemble de transitions. }
- 3 S : État { Variable intermédiaire }
- 4 AjouterÉtat(P_j, S) { Ajoute un état S à l'ensemble des états de P_j }
- 5 AjouterTransition($P_j, C.sj, Label(C.ti), S$) { Ajoute une transition $C.ti$ au LTS P_j dont l'état source est $C.sj$ et l'état cible est S }
- 6 AjouterTransitions($P_j, S, (C.sj) \bullet$) { Ajoute des transitions sortantes à l'état S à partir des transitions sortantes l'état $C.sj$ dans le LTS P_j }

FIG. 5.6 – Méthode de résolution de suppression d'acquittement

Le résultat de cette méthode est le LTS P_j qui inclut la résolution de l'incompatibilité C due à la suppression d'un acquittement. Un état intermédiaire S est ajouté au LTS P_j (voir la ligne 4). Des transitions, sont ajoutées au LTS P_j . Une première transition porte le libellé de l'opération d'acquittement supprimée (voir la ligne 5). D'autres transitions sortantes sont ajoutées à l'état intermédiaires (voir la ligne 6). Le LTS résultat décrit ainsi le comportement du médiateur qui sera déployé pour les clients adéquats.

5.3.3 Algorithme de résolution des incompatibilités

La résolution des incompatibilités qui font qu'une interface P' ne simule pas une autre interface P est basée sur le résultat de la détection de celles-ci. L'objectif de cette phase est de construire le LTS du médiateur. Le comportement du médiateur doit simuler celui de l'interface fournie P' donc le LTS du médiateur doit contenir les mêmes états et les mêmes transitions que le LTS de P' . Afin que le comportement du médiateur résolve les incompatibilités, son LTS est modifié par des ajouts d'états et de transitions selon chaque type d'incompatibilité détecté à des endroits précis du LTS du médiateur (voir par exemple les figures 5.2 et 5.3). Des expressions de résolution associées à chaque type d'incompatibilité sont utilisées pour déterminer les états et les transitions à ajouter. Cependant, cette résolution ne touche que les incompatibilités qui admettent l'utilisation d'une pile pour garder des informations existantes. En d'autres termes, si l'information n'existe pas, ni en envoi ni en réception, l'incompatibilité ne peut pas être résolue.

L'ensemble des incompatibilités qui peuvent être résolues automatiquement sont :

- Les incompatibilités comportementales induites par des changements dans la structure des messages (envois multiples/réception unique, décomposition de message, etc.).
- Les suppressions et les ajouts d'opérations d'acquittement.
- La suppression d'un terme d'une alternative avec l'hypothèse que le client peut choisir une autre option de l'alternative.

En particulier, les incompatibilités où des informations ne sont pas détenues par le médiateur, ne peuvent être résolues automatiquement. Dans ce cas, notre approche se contente de retourner ces incompatibilités pour qu'ils soient résolus de manière manuelle avec l'intervention des concepteurs des applications clients qui utilisent le service.

```

1 Résolution (ensChange : { Change }; Pi : LTS; Pj : LTS ) : LTS
2 { Résolution(ensChange,Pi,Pj) : la détection des incompatibilités est réalisée sur les deux au-
  tomates Pi et Pj où l'ensemble des incompatibilités élémentaires est ensChange. Le résultat
  retourné est le LTS du médiateur. }
3 ltsRes : LTS { variable résultat }
4 RE : {resolutionExp}
5 rEx : resolutionExp
6 ltsRes ← Pj
7 RE ← ChargerRE()
8   Pour tout rEx ∈ RE
9     Évaluer(rEx,Pi,Pj,ltsRes)
10 Retourner(ltsRes)

```

FIG. 5.7 – Algorithme de résolution des incompatibilités

Dans l'algorithme, illustré dans la figure 5.7, les expressions de résolutions se limitent aux incompatibilités résolvable automatiquement.

Le LTS résultat de l'algorithme est le LTS *ltsRes*. Il est initialisé avec le LTS de l'interface *Pj* (voir les lignes 3 et 6). Les expressions des méthodes de résolution sont utilisées (voir les lignes 4 et 7). Ainsi, pour chaque expression de résolution, la méthode associée est exécutée pour effectuer les modifications appropriées au *ltsRes* (voir la ligne 9).

5.4 Génération automatique de médiateurs

Dans cette section, nous introduisons la génération automatique de médiateurs telle qu'elle est proposée dans le canevas *ArchiMed*. Pour cela, nous présentons le mécanisme des interactions entre les clients et le service via des médiateurs dans la section 5.4.1. Puis, dans la section 5.4.2, nous détaillons le mécanisme de réutilisation de médiateurs. Enfin, dans la section 5.4.3 nous définissons une mesure de similarité entre les interfaces qui permet de sélectionner une interface conforme avec l'interface requise d'un client.

5.4.1 Interactions via des médiateurs

La figure 5.8 présente les interactions entre les services et les clients via des médiateurs. Une flèche unidirectionnelle désigne l'envoi de messages de la source de la flèche vers la destination de la flèche. La flèche bidirectionnelle désigne que les échanges de messages se font dans les deux sens entre les éléments reliés par cette flèche. Les fonctions offertes s'articulent en deux phases, selon qu'elles sont exécutées à la conception d'une nouvelle version d'interface, ou à l'exécution d'une conversation initiée par un client.

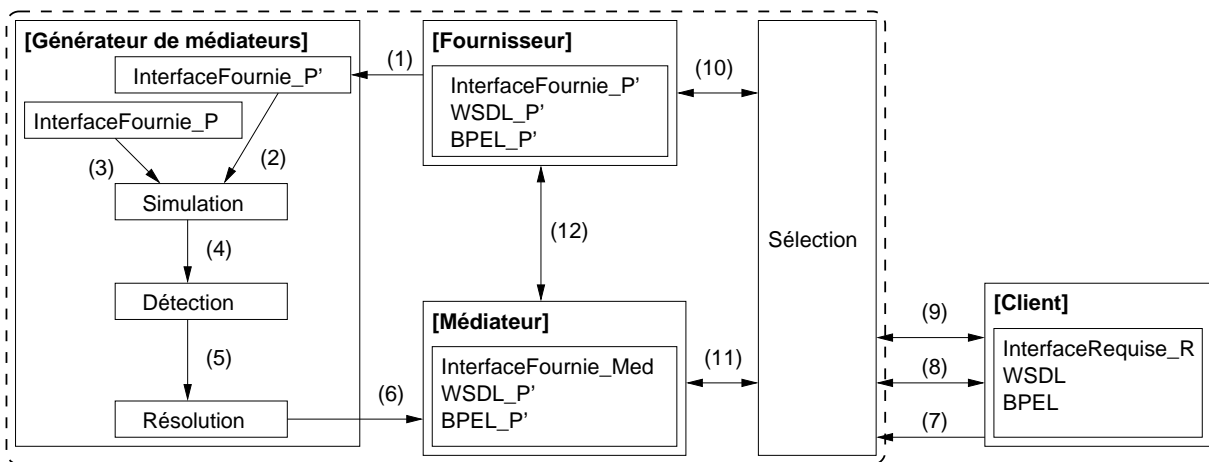


FIG. 5.8 – Interactions via des médiateurs

La première phase est sous le contrôle du *générateur de médiateurs* (voir les étapes (1) à (6) dans la figure 5.8). Le générateur s'appuie sur les versions P et P' de l'interface fournie (voir (1)) du service fournisseur. Sur la base des représentations en automates des versions P et P' (voir (2) et (3)), le module *Simulation* vérifie si la nouvelle interface P' simule P . Le résultat de ce calcul (voir (4)) est transmis au module de détection d'incompatibilités qui retourne les incompatibilités associés à des différences dans les définitions des interface de P et de P' qui font que P' ne simule pas P . En partant des résultats des incompatibilités détectées (voir (5)), un autre module les résout en

construisant et en déployant le médiateur correspondant (voir (6)).

Ensuite, lorsqu'un client initie une conversation avec le service (voir (7)), une première étape de sélection permet d'étudier, après l'analyse de l'interface exposée par le client, la compatibilité de cette dernière avec la version courante P' de l'interface du service. Il peut en découler trois situations différentes :

1. L'interface du client est conforme à la version actuelle de l'interface fournie. Dans ce cas les interactions entre le client et le fournisseur ne transitent par aucun médiateur (voir (9) et (10)).
2. L'interface du client n'est pas conforme avec la version actuelle, et n'est conforme avec aucune version précédente. Dans ce cas il n'existe aucun médiateur pour ce client. La conversation est interrompue dès son initiation car elle ne peut pas aboutir.
3. Il existe une version conforme à l'interface du client, et celle-ci est unique (ce résultat est démontré dans la section 5.4.2). Les interactions sont gérées par le médiateur associé (voir (9), (11) et (12)).

La sélection du médiateur s'appuie sur les résultats d'une mesure de similarité qui est détaillée plus loin dans la section 5.4.3.

5.4.2 Utilisation de médiateurs

Dans notre approche, la conformité entre l'interface requise d'un client et une nouvelle définition d'une interface fournie est garantie par l'introduction d'un médiateur. Ainsi, dans le cas où une interface requise d'un client C_1 est conforme avec une interface fournie P_1 et que P_1 évolue vers P_2 , un médiateur $M_{1,2}$ est introduit pour garantir la conformité entre C_1 et la nouvelle version d'interface fournie P_2 . Cependant, si l'interface fournie P_2 évolue vers P_3 , le médiateur $M_{1,2}$ n'est plus nécessaire car il est remplacé par un autre médiateur $M_{1,3}$ qui garantit la conformité entre l'interface requise C_1 et la nouvelle version d'interface fournie P_3 . Pour l'interface requise d'un nouveau client C_2 qui était conforme à la précédente version d'interface fournie P_2 , un autre médiateur $M_{2,3}$ est généré pour garantir la conformité entre C_2 et la nouvelle version P_3 . Ce mécanisme de génération et de suppression de médiateur est généralisé à des évolutions d'une interface fournie vers N versions (voir Figure 5.9).

Lors de la définition de la n ème version de l'interface fournie P_n , le client d'interface requise C_1 (qui était conforme à l'interface fournie précédente P_{n-1} par le biais du médiateur $M_{1,n-1}$) est conforme avec cette nouvelle version grâce à un nouveau médiateur $M_{1,n}$. Les indices de valeurs inférieures désignent des versions d'interface antérieures à des versions d'interfaces d'indices de valeurs supérieures. Le précédent médiateur ($M_{1,n-1}$)

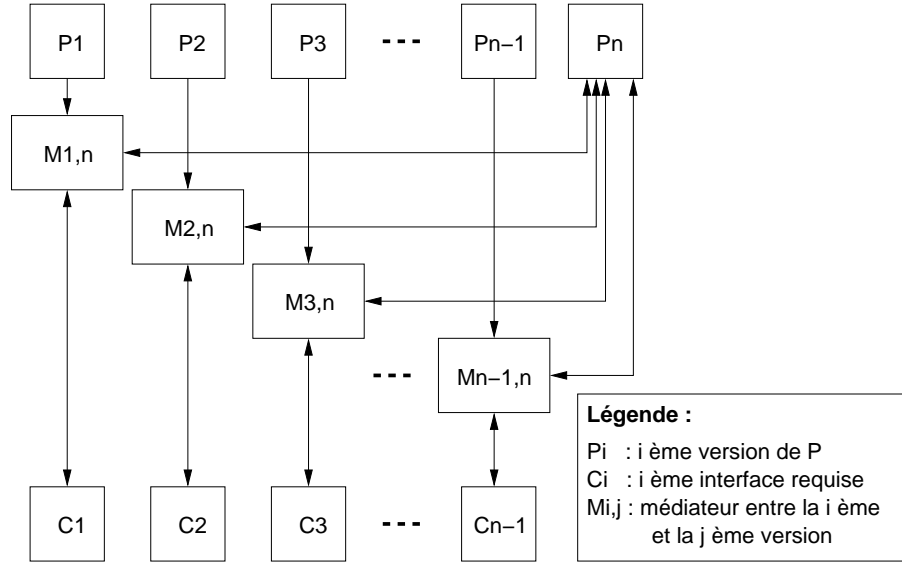


FIG. 5.9 – Génération de médiateurs pour plusieurs interfaces.

n'étant plus utile, il est supprimé. Pour garantir la conformité des $N - 1$ interfaces requises des clients avec la nouvelle version P_n , les $n - 2$ médiateurs qui garantissaient la conformité avec la précédente version P_{n-1} sont supprimés pour être remplacés par $n - 1$ nouveaux médiateurs qui assurent la conformité des interfaces requises avec la nouvelle version P_n .

Unicité du médiateur

Si un médiateur est conforme avec une interface requise alors il est unique. Pour démontrer ce résultat, un raisonnement par l'absurde est utilisé où l'hypothèse de l'existence de deux médiateurs distincts conformes à une même interface requise est prise. Soient P_i, P_j, P_r trois LTSs des versions d'interface tel que $i < j < r$ et que $P_i \not\leq P_j \wedge P_j \not\leq P_r \wedge P_i \not\leq P_r$. Soit P_r la version actuelle, les médiateurs $M_{i,r}$ (entre P_i et P_r) et $M_{j,r}$ (entre P_j et P_r) sont générés. Supposons qu'il existe une interface requise R telle que $R \sim M_{i,r} \wedge R \sim M_{j,r}$ (i.e. $\bar{R} \preceq M_{i,r} \wedge \bar{R} \preceq M_{j,r}$). D'après la définition d'un médiateur, son comportement inclut les comportements des deux versions qu'il gère d'où : $M_{i,r} \equiv P_i \cup P_r$ et $M_{j,r} \equiv P_j \cup P_r$ (où \equiv désigne l'équivalence ou la bissimulation). Ainsi, nous avons $\bar{R} \preceq P_i \cup P_r \wedge \bar{R} \preceq P_j \cup P_r$. De cette expression est déduit : $\bar{R} \preceq P_i \wedge \bar{R} \preceq P_j$. Ce qui revient à dire que P_i et P_j sont équivalents dans le contexte R (i.e. $P_i \equiv_R P_j$). Cette équivalence implique que $P_i \preceq_R P_j \wedge P_j \preceq_R P_i$. Or, $P_i \not\leq P_j$. Cette démonstration est récapitulée ci-après.

Hypothèses :

- (1) : P_i, P_j, P_r des LTSs telle que $i < j < r$
 { P_r est la version actuelle, P_j est une version antérieure à P_r , P_i est une version antérieure à P_r . }
 (2) : $P_i \not\leq P_j \wedge P_j \not\leq P_r \wedge P_i \not\leq P_r$
 { P_j ne simule pas P_i , P_r ne simule pas P_j et P_r ne simule pas P_i }
 (3) : $M_{i,r}$ un médiateur entre P_i et P_r et $M_{j,r}$ un médiateur entre P_j et P_r
 (4) : R une interface requise telle que $R \sim M_{i,r} \wedge R \sim M_{j,r}$ { Hypothèse inverse. }
 { R est conforme à $M_{i,r}$ et à $M_{j,r}$ }

Construction de la contradiction :

- (5) : (3) $\Rightarrow M_{i,r} \equiv P_i \cup P_r \wedge M_{j,r} \equiv P_j \cup P_r$
 { D'après la définition du comportement d'un médiateur qui inclut les comportements des deux versions. }
 (6) : (4) $\Rightarrow \bar{R} \preceq M_{i,r} \wedge \bar{R} \preceq M_{j,r}$
 (7) : (5) \wedge (6) $\Rightarrow \bar{R} \preceq P_i \cup P_r \wedge \bar{R} \preceq P_j \cup P_r$
 { Par transitivité de la relation de simulation et de la relation d'équivalence. }
 (8) : (7) $\Rightarrow \bar{R} \preceq P_i \wedge \bar{R} \preceq P_j$
 $\Rightarrow P_i \equiv_R P_j$
 $\Rightarrow P_i \preceq_R P_j \wedge P_j \preceq_R P_i$
 Or d'après (2), $P_i \not\leq P_j$. { Contradiction avec $P_i \preceq_R P_j$. }

Réutilisation de médiateurs

Dans la succession des évolutions d'une interface fournie vers de nouvelles versions, une évolution peut produire à une version d'interface fournie qui est équivalente ou bien qui simule une interface d'une version antérieure. Les médiateurs qui étaient déployés pour cette version antérieure peuvent être réutilisés avec cette version courante d'interface fournie.

Soient P_i, P_k et P_r trois versions d'interfaces fournies telle que :

- $i < k$ et $k < r$
- P_r est la version actuelle
- $P_i \not\leq P_k$
- $P_k \preceq P_r$ (la version actuelle P_r simule la version précédente P_k)

Avant d'évoluer vers la version actuelle P_r , l'interface fournie dans sa version P_k était utilisée par des clients dont l'interface requise est conforme à l'interface P_i par le biais d'un médiateur $M_{i,k}$, d'où $P_k \preceq M_{i,k}$. Étant donné que $P_k \preceq P_r$ alors $M_{i,k} \equiv_{P_k} P_r$. De ce fait, le médiateur $M_{k,r}$ à déployer entre les versions P_k et P_r est $M_{i,k} \equiv M_{k,r}$.

Ainsi, avant la génération des médiateurs, il faut tester la simulation entre la nouvelle version d'interface avec toutes les versions antérieures. Si le résultat est positif avec une version antérieure d'interface P_i alors :

- le client C_i dont l'interface requise est conforme avec la version d'interface fournie P_i va interagir directement avec la version actuelle de l'interface fournie P_r .
- et chaque client C_h dont l'interface requise est conforme à une version d'interface fournie P_h (telle que $h < i$ et que $P_h \not\subseteq P_i$) va interagir avec la nouvelle version en s'appuyant sur un médiateur précédemment généré $M_{h,i}$ et qui sera de nouveau déployé.

5.4.3 Sélection des médiateurs

Dans le canevas *ArchiMed*, le médiateur qui doit conduire la conversation entre un client et le service, est sélectionné. La sélection du médiateur est basée sur l'utilisation d'une mesure de similarité des interfaces. L'objectif est de trouver l'interface dont le comportement est le plus proche ou équivalent de la définition de l'interface fournie sollicité par le client. L'interface recherchée est comparée à toutes les autres définition de l'interface du service. Les interfaces les plus similaires à l'interface recherchée sont celles qui engendrent le moins d'incompatibilités.

Dans la détection des incompatibilités, l'algorithme compare les deux automates P_i et P_j qui représente respectivement, une version antérieure et une version actuelle de l'interface fournie. Une incompatibilité n'est détectée que lorsque le comportement de la version actuelle ne simule le comportement de la version antérieure. Pour rappel, l'algorithme de détection retourne un ensemble de tuples *ensRes* d'incompatibilité où chaque nuplet est de la forme $\langle si, ti, sj, tj \rangle$. Tandis que si and sj représentent les états de P_i et P_j respectivement, ti et tj sont soit des valeurs nulles soit des transitions sortantes de si et sj respectivement.

Au lieu de retourner uniquement un nombre d'incompatibilités entre deux interfaces comportementales des services, nous proposons une mesure de similarité. Cette mesure indique jusqu'à quel point le comportement décrit dans une interface P_j peut simuler le comportement décrit dans une autre interface P_i . Les valeurs de cette mesure sont comprises entre zéro et un. La valeur zéro (respectivement un) signifie que l'interface P_j ne simule pas (respectivement simule) l'interface P_i . Les valeurs comprise entre zéro et un désigne que l'interface P_j peut simuler l'interface P_i à condition d'effectuer certaines modifications dans l'interface P_j . Plus la valeur est proche de un et moins il y aura de modifications à apporter dans l'interface P_j .

Score de la similarité d'une paire d'états

Dans l'algorithme de détection des incompatibilités élémentaires, les deux LTSs P_i et P_j sont parcouru de manière synchrone. Un ensemble des paires d'états déjà visitées est obtenu tel que $visitées \subseteq S_i \times S_j$ et S_i (respectivement S_j) est l'ensemble des états de P_i (respectivement P_j). Pour chaque paire d'états $(si, sj) \in visitées$, le degré de simulation d'une paire d'états $(si, sj) \in visitées$ est calculé par la fonction Wqs qui retourne des valeurs dans l'intervalle $[0, 1]$ (voir l'annexe C pour la démonstration de ce résultat). La fonction $Wqs : S_i \times S_j \rightarrow [0, 1]$, où S_i est l'ensemble des état de P_i et S_j est l'ensemble des états de P_j , est définie comme suit :

$$Wqs((si, sj)) = \begin{cases} 1 & \text{si, } si \bullet = \{\} \\ \frac{\|Diff((si, sj))\|}{\sum_{i=1} Weight(D_i) + \|Label(si \bullet) \cap Label(sj \bullet)\|} & \text{sinon} \end{cases} \quad (5.1)$$

D'après la fonction ci-avant, si l'état si ne contient pas de transitions sortantes alors le score de similarité le plus haut est attribué à la paire d'états. $Diff((si, sj))$ est un ensemble des incompatibilités détectées au niveau de la paire d'états (si, sj) tel que $Diff((si, sj)) \subseteq ensRes$, et $D_i \in Diff((si, sj))$ pour tout indice $i = 1.. \|Diff((si, sj))\|$. La fonction $Weight$ retourne la valeur d'une pénalité associée à chaque type d'incompatibilité, et $0 \leq Weight(D_i) < 1$. La somme de toutes les pénalités est additionnée au nombre de libellés en commun des transitions sortantes des deux états si et sj . Les libellés en commun des transitions sortantes de de si et de sj désignent le cas où aucune incompatibilité n'est détectée. Ainsi, la plus grande valeur de similarité est attribuée aux libellés en commun (voir (5.1) : $\|Label(si \bullet) \cap Label(sj \bullet)\|$). Afin de calculer le score de similarité d'une paire d'états, la somme des pénalités et du score des libellés en commun est divisée par la somme du nombre des incompatibilités et du nombre de libellés en commun des transitions sortantes de si et de sj (voir (4.1) : $\|Diff((si, sj))\| + \|Label(si \bullet) \cap Label(sj \bullet)\|$). Par exemple, dans la figure 3.3 de la section 3.3.1, si la valeur de la pénalité pour une suppression d'opération est de 0.5, alors le score de similarité pour la paire d'états $(CommandeHorsLigne, CommandeHorsLigne)$ est : $Wqs((CommandeHorsLigne, CommandeHorsLigne)) = \frac{0.5+1}{1+1} = 0.75$.

Mesure de similarité moyenne

Après avoir calculé le score de similarité de toutes les paires d'états visitées, la mesure de similarité moyenne de P_i et de P_j est donnée par la fonction Mqs qui retourne des valeurs dans l'intervalle $[0, 1]$ (voir l'annexe C pour la démonstration de ce résultat). La fonction Mqs est définie comme suit :

$$Mqs(P_i, P_j) = \frac{\sum_{i=1}^{\|visitées\|} Wqs(PS_i)}{\|visitées\|} \quad (5.2)$$

PS_i est une paire d'états tel que $PS_i \in visitées$ pour tout $i = 1.. \|visitées\|$. La mesure de similarité entre une interface P_i et une autre interface P_j est la moyenne des mesures de similarités des paires d'états visitées lors de la détection des incompatibilités (voir 5.2). La somme des mesures de similarité des paires d'états $\sum_{i=1}^{\|visitées\|} Wqs(PS_i)$ est divisée par le nombre des paires d'états visité $\|visitées\|$. Dans l'exemple de la figure 3.3 de la section 3.3.1, si les valeurs des pénalités d'ajout, de suppression et de modification sont toutes 0.5, alors la mesure de similarité moyenne de P_i et de P_j est : $Mqs(P_i, P_j) = 0.875$.

5.5 Conclusion

La détection des incompatibilités est un mécanisme complètement automatisé et permet ainsi de rendre compte de toutes les incompatibilités élémentaires. La résolution des incompatibilités qui font qu'une interface P' ne simule pas une autre interface P est réalisé par des médiateurs qui sont automatiquement générés. Le comportement du médiateur est défini dans un LTS. Ce dernier est obtenu en reproduisant le comportement de l'interface P' à laquelle sont ajoutés les adaptations nécessaires pour résoudre les incompatibilités. La résolution des incompatibilités est un mécanisme qui prend en compte plusieurs définitions de l'interface d'un service. Ainsi, plusieurs médiateurs sont générés pour prendre en compte les conversations de tous les clients qui sollicitent les différentes définitions de l'interface du service. Les conversations entre le client et le service s'effectuent par l'intermédiaire du médiateur. La sélection du médiateur est réalisée par une mesure de similarité entre la définition de l'interface fournie sollicitée par le client et la définition courante de l'interface du service. La mise en œuvre de la détection et de la résolution des incompatibilités dans le canevas *ArchiMed* sont présentées dans le chapitre 6.

Chapitre 6

Mise en œuvre d'ArchiMed

Sommaire

6.1	Introduction	117
6.2	Mise en œuvre du prototype	118
6.2.1	Architecture logicielle	118
6.2.2	Détails d'implantation	119
6.2.3	Diagramme de classes de la détection des incompatibilités	122
6.3	Scénarios d'utilisation de ArchiMed	123
6.3.1	Point d'entrée de ArchiMed	123
6.3.2	Détection des incompatibilités entre deux interfaces	124
6.3.3	Mesure de similarité entre les interfaces	126
6.3.4	Détection des incompatibilités automatiquement résolubles	127
6.4	Tests sur la détection des incompatibilités	128
6.4.1	Construction de la base de tests	129
6.4.2	Tests de similarité des interfaces	129
6.4.3	Etudes comparatives	130
6.5	Conclusion	134

6.1 Introduction

La solution que nous proposons, pour la détection et la résolution des incompatibilités est mise en œuvre dans le canevas *ArchiMed*. Dans la section 6.2, nous détaillons les différents modules de l'architecture du canevas. Les scénarios d'utilisation du canevas illustrent les différentes fonctionnalités de détection des incompatibilités (voir la section 6.3). Enfin, dans la section 6.4, nous présentons une validation expérimentale du canevas *ArchiMed* ainsi qu'une étude quantitative comparative à des travaux similaires.

6.2 Mise en œuvre du prototype

Dans cette section, nous présentons les différents composants de l'architecture logicielle (voir section 6.2.1), ainsi que les détails de la mise en œuvre de la solution de détection et de résolution des incompatibilités (voir section 6.2.2). Le diagramme des classes qui mettent en œuvre la détection des incompatibilités est détaillé dans la section 6.2.3.

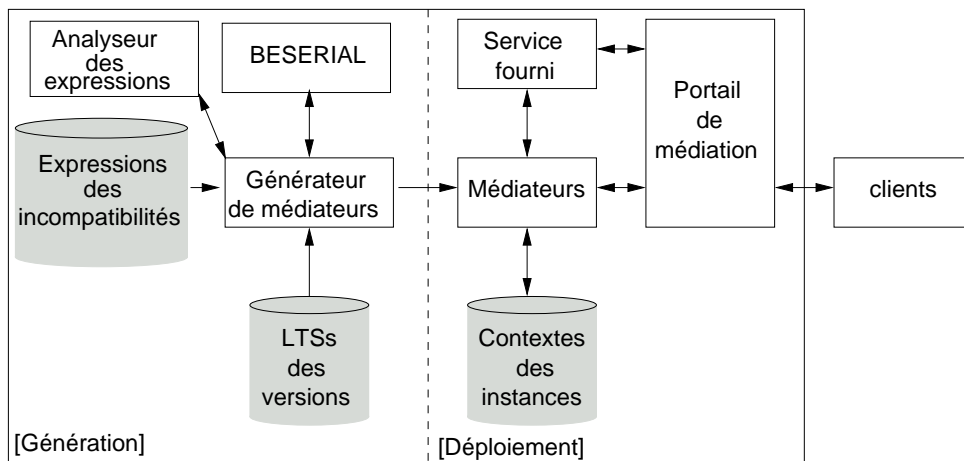
6.2.1 Architecture logicielle

La figure 6.1 schématise l'architecture du canevas ArchiMed. Nous prenons pour hypothèse que celle-ci est installée chez un fournisseur de services. Pour chaque service exposé par le fournisseur, les modules de la partie déploiement doivent être déployés. Ils sont exécutés chaque fois qu'un client initie une conversation avec le service. A contrario, les modules de la partie Génération ne sont pas spécifiques à un service donné. Ils sont exécutés dès lors que la définition de l'interface d'un service évolue.

Les modules pour la génération sont **BESERIAL** pour la détection des incompatibilités, l'analyseur des expressions de détection des incompatibilités automatiquement résolubles et le générateur de médiateurs (voir la figure 6.1, partie [Génération]). Les modules pour la gestion des interactions avec les services sont, les médiateurs générés et le portail de médiation (voir la figure 6.1, partie [Déploiement]).

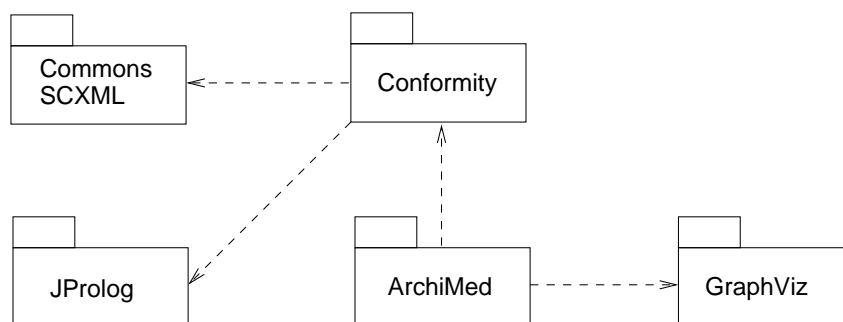
Le générateur de médiateurs s'appuie sur l'historique des versions de l'interface fournie. Ces versions sont stockées sous forme de LTSs pour être comparées à la dernière version de l'interface fournie. Lors de la comparaison, la détection des incompatibilités est réalisée en sollicitant le compilateur qui évalue les expressions associées à chaque incompatibilité. Le module **Analyseur des expressions** évalue les expressions associées à chaque cas d'incompatibilité puis extrait de l'ensemble des incompatibilités détectées celles qui peuvent être résolues automatiquement.

Le module **Générateur de médiateur** construit automatiquement les médiateurs à partir des LTSs des versions du service et des résultats de la détection des incompatibilités. Les médiateurs qui assurent la conformité entre les différentes interfaces requises et la nouvelle interface fournie sont déployés par le générateur de médiateurs (voir [Déploiement]). Lorsqu'un client initie une conversation, le **portail de médiation** vérifie la conformité de l'interface requise par ce client et détermine si un médiateur est nécessaire. Dans le cas où la conformité n'est pas garantie (ni par un médiateur ni par le service actuel), le portail de médiation interrompt la conversation avec le client dès son initialisation car les incompatibilités ne pourront pas être résolues.

FIG. 6.1 – Architecture du canevas *ArchiMed*

6.2.2 Détails d'implantation

La mise en œuvre du canevas *ArchiMed* est faite en *Java*. Elle s'appuie sur l'utilisation d'autres *APIs Java* définie dans d'autres projets. La figure 6.2 illustre les différents paquetages du canevas ainsi que les dépendances entre ces derniers. Le paquetage *ArchiMed* correspond au point d'entrée dans l'architecture. Il utilise les autres paquetages décrits plus bas.

FIG. 6.2 – Diagramme des paquetages du canevas *ArchiMed*

Le principal paquetage du canevas est le paquetage **Conformity**. Il contient les classes qui permettent de définir les interfaces comportementales des services et les classes qui construisent les résultats de la détection des incompatibilités. Le paquetage **Conformity** dépend de deux autres paquetages (**CommonsSCXML** et **JProlog**) qui sont explicité ci-après.

La représentation des automates (CommonsSCXML)

La représentation en automates des interfaces des services s'appuie sur le standard SCXML⁷ (*State Chart XML*), basé sur XML. SCXML s'appuie sur les définitions des automates données par Harel [39].

Afin de rendre exécutables des automates représentés en SCXML, le groupe *Apache* propose une API Java : *Commons SCXML*⁸. Ainsi, en partant d'une représentation XML d'un automate, l'API interprète le document pour initialiser une classe qui est le moteur d'exécution de l'automate. Ce moteur offre quelques primitives pour l'exécution de l'automate : déclencher une transition, vérifier si un état est final ou pas, etc.

Afin d'adapter les représentations des automates dans le format SCXML à notre définition des LTSs, nous avons ajouté des étiquettes d'envois et de réceptions de messages ainsi que l'identifiant de l'instance du LTS, dans les attributs événements des transitions. Le document XML donné dans la figure 6.3 illustre la représentation dans le format SCXML de l'état de confirmation. Ce dernier contient une transition dont le libellé est un événement `event` et un état cible `target` qui est `facture`. Le libellé de la transition est composé de trois parties. La première partie est l'identifiant de la conversation, dans ce cas il s'agit `idC01`. La deuxième partie `envoi` désigne que l'opération *Confirm* est une opération d'envoi. En d'autres termes, lorsque l'automate atteint l'état `confirmation`, le moteur d'exécution de l'automate déclenche un événement d'envoi de confirmation dans la conversation identifiée par `idC01`. Après le déclenchement de l'événement, l'état courant de l'automate passe à l'état cible de la transition qui est `facture`.

```
<state id="confirmation">
  <transition event="idC01.envoi.Confirm" target="facture"/>
</state>
```

FIG. 6.3 – Représentation des automates dans le format SCXML

Chaque document SCXML décrivant le contexte d'exécution d'instance de conversation est généré à partir du moteur d'exécution. Ce document contient l'identifiant de la conversation à laquelle il est associé. Ce document est sauvegardé dans une base de données (c'est le contexte) puis il est rechargé au moment opportun pour reprendre l'exécution du processus.

⁷<http://www.w3.org/TR/scxml/>

⁸<http://commons.apache.org/scxml/>

Le compilateur Prolog (JProlog)

Afin d'interpréter et d'évaluer les expressions de cas des incompatibilités résolubles, nous avons utilisé un compilateur Prolog⁹ implanté en Java. La base de faits reflète les descriptions des deux automates comparés mais également les incompatibilités détectées. La base de règles contient les règles associées à chaque cas d'incompatibilité résoluble. Ces incompatibilités sont extraites automatiquement grâce au compilateur Prolog qui construit le sous-ensemble résultat des cas d'incompatibilités résolubles.

Pour étendre la base de règles, il suffit d'ajouter l'expression qui modélise un nouveau cas d'incompatibilité qui peut être résolu automatiquement.

Le compilateur Java pour Prolog est utilisé et est intégré au canevas *ArchiMed*. La base de faits contient les descriptions des deux LTSs à comparer. En d'autres termes, les faits assignés par le prédicat $Transition(s1, s2, t)$ indiquent toutes les transitions des deux automates où : $s1$ est l'état source de la transition t et $s2$ est l'état cible. D'autres faits indiquent les états initiaux et les états finals. Les faits des incompatibilités détectées sont assignés par le prédicat $basic_change(s1, t1, s2, t2, change)$ où :

- $s1$ est un état du LTS de la précédente version et $t1$ une de ses transitions sortantes.
- $s2$ est un état du LTS de la version actuelle et $t2$ une de ses transitions sortantes.
- $change$ est le type de changement de base (ajout, suppression ou modification).

Une fois la base de connaissances Prolog chargée, elle peut être interrogée par des requêtes pour tester chacune des incompatibilités qui peuvent être résolues automatiquement.

Représentation graphique des incompatibilités (GraphViz)

Dans la canevas *ArchiMed*, les incompatibilités sont retournées de manière visuelle à l'aide de graphes. Ces graphes sont les deux définitions des automates comparés sur lesquels sont indiqués les transitions et les états concernés par un ou plusieurs types d'incompatibilité. Ces résultats servent de supports visuels pour les concepteurs des applications clientes dont les interfaces requises sont incompatibles avec la nouvelle définition de l'interface fournie.

Pour réaliser ces graphes, *ArchiMed* fournit un fichier textuel qui contient la description des deux automates ainsi que des différences entre ces deux automates. Ce fichier est interprété par un outil externe *Graphviz - Graph Visualization Software*¹⁰ qui retourne une image du graphe dans le format souhaité.

⁹<http://jlogic.sourceforge.net/>

¹⁰<http://www.graphviz.org/>

6.2.3 Diagramme de classes de la détection des incompatibilités

La figure 6.4 illustre une partie du diagramme des classes du canevas *archiMed*. Ces différentes classes représentent les interfaces comportementales des services par des automates ainsi que les algorithmes de détection des incompatibilités.

Les classes `AbstractStateMahine`, `State` et `Transition` sont définies dans le paquetage `CommonsSCXML`. Les classes `LTSInterface`, `IncompatibilityDetection`, `TupleRes`, `GnuPlot` et `DotFile` sont définies dans le paquetage `Comformity`.

Les algorithmes de détection des incompatibilités sont mis en œuvre dans la classe `IncompatibilityDetection`. Cette classe est reliée à deux interfaces représentées par des LTS (les deux LTSs à comparer). La classe `LTSInterface` est une sous-classe de la classe `AbstractStateMahine`.

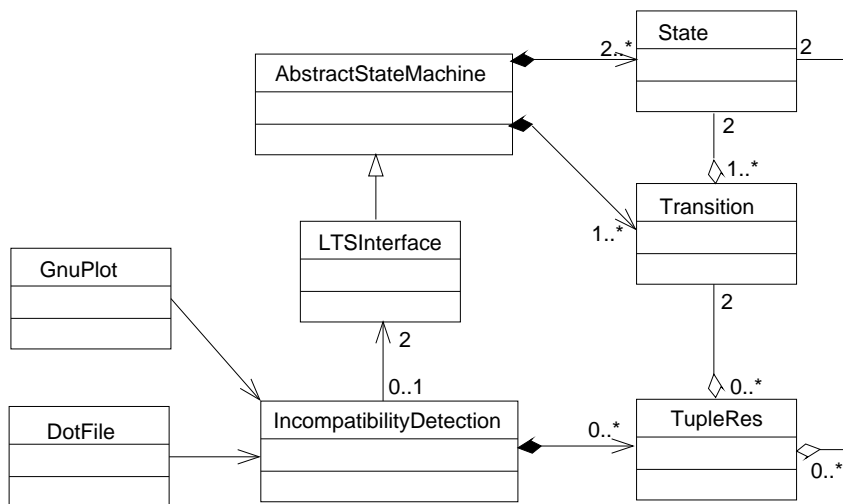


FIG. 6.4 – Diagramme des classes de la détection

La classe `AbstractStateMahine` qui modélise les automates est reliée à deux classes qui sont : i) la classe `State` qui représente un état, et ii) la classe `Transition` qui représente une transition.

Une transition est définie par deux états, un état source et un état cible. Ceci est traduit par une multiplicité égale à deux dans l'association qui relie la classe `State` à la classe `Transition`. Le résultat de la détection des incompatibilités est un ensemble de nuplets. Ces nuplets sont représentés par la classe `TupleRes`. Cette classe est liée à la classe `State` (les états où l'incompatibilité est détectée) et à la classe `Transition` (les transition concernées par l'incompatibilité). Les deux classes `GnuPlot` et `DotFile` permettent de générer des représentations graphiques des résultats de la détection.

6.3 Scénarios d'utilisation de ArchiMed

Dans cette section nous présentons des scénarios d'utilisation des différentes fonctionnalités du canevas *ArchiMed* [2]. Dans la section 6.3.1, nous introduisons de manière globale le point d'entrée du canevas. La section 6.3.2 présente les fonctionnalités de détection des incompatibilités ainsi que les modes d'affichages des résultats de la détection. L'utilisation de la fonctionnalité de mesure de similarité entre les interfaces est détaillée dans la section 6.3.3. Enfin, un scénario d'utilisation de la détection des incompatibilités automatiquement résolubles est décrit dans la section 6.3.4.

6.3.1 Point d'entrée de ArchiMed

La mise en œuvre de *ArchiMed* se présente comme une archive *Java* qui est *archiMed.jar*. Dans le répertoire courant de *archiMed.jar* il y a :

- un répertoire *lib* qui contient toutes les *API* sur lesquelles s'appuie le canevas.
- un fichier *save.plt* qui contient la définition des paramètres de configuration *gnuplot* (outil d'affichage des graphes).

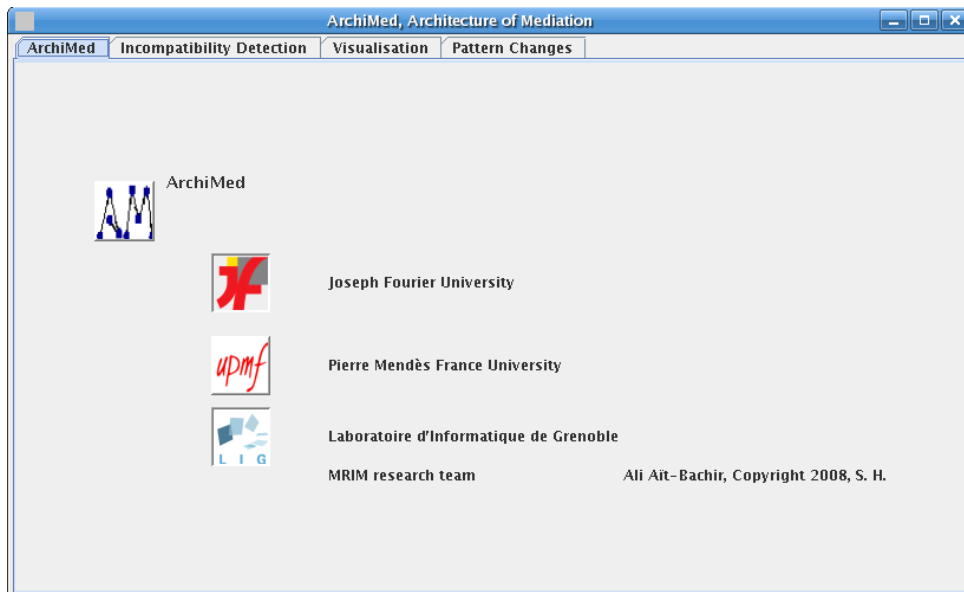


FIG. 6.5 – *ArchiMed* : vue globale du canevas

La figure 6.5 illustre une vue globale du canevas. L'interface graphique se décline en quatre parties qui sont chacune associée à un anglet de l'interface :

- ArchiMed : accueil du canevas,
- Incompatibility detection : détection des incompatibilités et calcul de la mesure de similarité,
- Visualisation : résultats graphiques de la détection des incompatibilités et du calcul des mesures de similarité,
- Pattern changes : détection des cas d'incompatibilités qui peuvent être résolues automatiquement.

Une démonstration du canevas est accessible en ligne à l'adresse :

<http://www-clips.imag.fr/mrim/User/ali.ait-bachir/webServices/webServices.html>

Dans les sections 6.3.2 à 6.3.4 qui suivent nous détaillons chacune des parties du canevas *ArchiMed*.

6.3.2 Détection des incompatibilités entre deux interfaces

Le canevas *ArchiMed* permet de réaliser les étapes des tests de simulation et de détection des incompatibilités entre les interfaces des services web décrites dans le format SCXML.

La figure 6.6 illustre une capture d'écran de la partie Incompatibility detection.

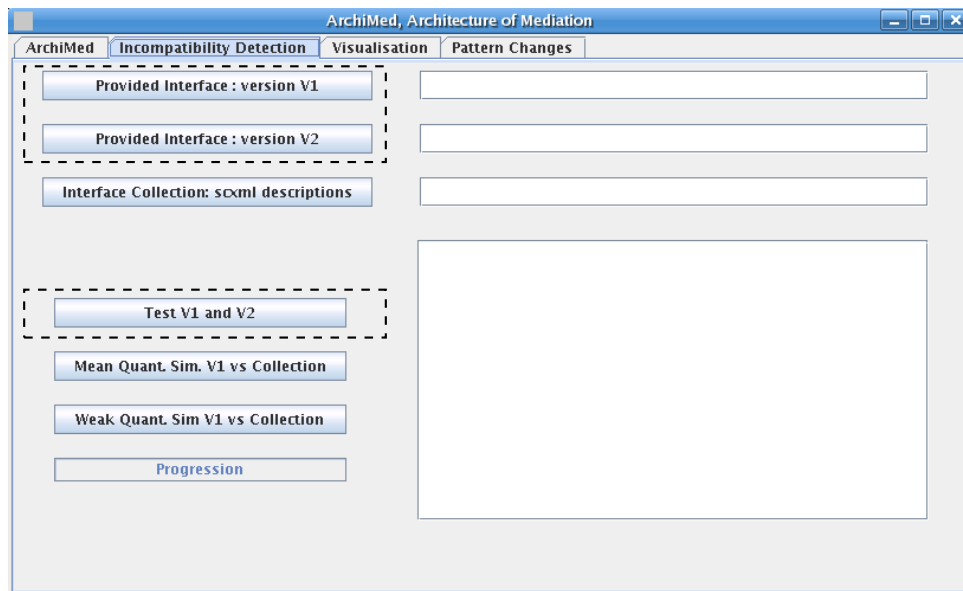


FIG. 6.6 – Détection des incompatibilités

Les deux boutons **Provided interface : version V1** et **Provided interface : version V2** permettent de sélectionner les fichiers en format SCXML qui contiennent les descriptions de deux interfaces fournies. Le bouton **Test V1 and V2** permet de lancer la détection des incompatibilités qui font que l'interface V2 ne simule pas l'interface V1. Le résultat graphique de la détection est un fichier `v1v2.dot` généré automatiquement et il sera créé dans le répertoire spécifié par l'utilisateur. Ce fichier contient une représentation textuelle des deux LTSs ainsi que des incompatibilités qui y sont détectées.

La figure 6.6 illustre un exemple de détection des incompatibilités entre une interface décrite dans le fichier `orderManagement-6-1-0.xml` et une autre interface décrite le fichier `orderManagement-6-1-0.xml`. Les exemples sont des descriptions de processus de commande d'articles offerts par des services.

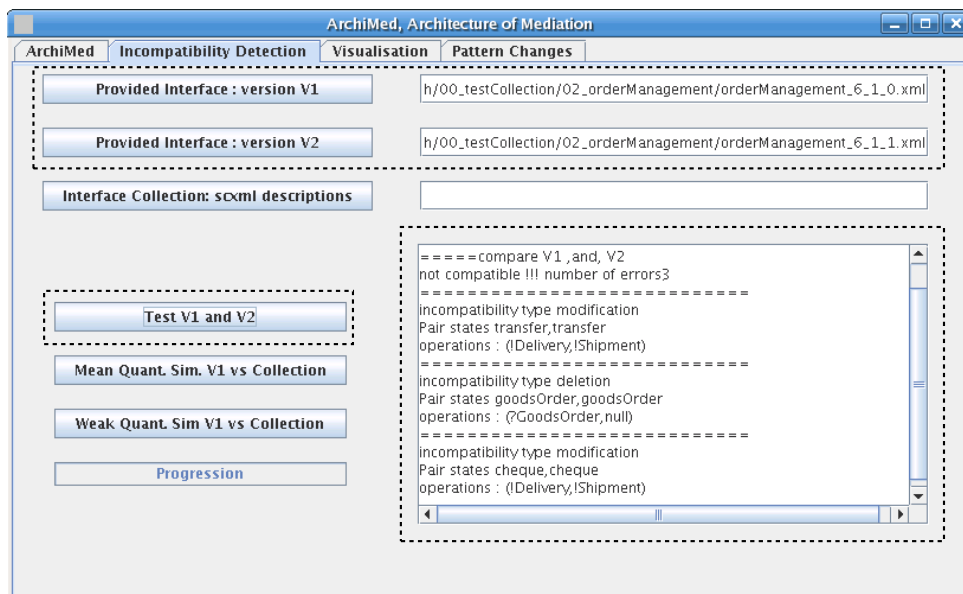


FIG. 6.7 – Résultats sous forme textuelle de la détection des incompatibilités

Le résultat de la détection est retourné dans une zone de texte. Ici, les incompatibilités détectées sont au nombre de trois (deux modifications et une suppression d'opérations). Chaque incompatibilité est localisée dans les états des deux automates comparés.

Une fois les incompatibilités détectées, il est possible de retourner un résultat graphique aux utilisateurs du canevas. Dans la figure 6.8, le bouton **Select the .dot file** permet de sélectionner le fichier qui contient la description du graphe du résultat de la détection.

Après avoir sélectionné le fichier `.dot`, le répertoire où l'image `.ps` doit être sauvegardée est sélectionné en utilisant le bouton **Generate the .ps file**. La représentation graphique des deux automates ainsi que les incompatibilités détectées sont affichées ici dans un document

PostScript que l'utilisateur peut visualiser.

La figure 6.8 illustre le résultat graphique de la détection des incompatibilités.

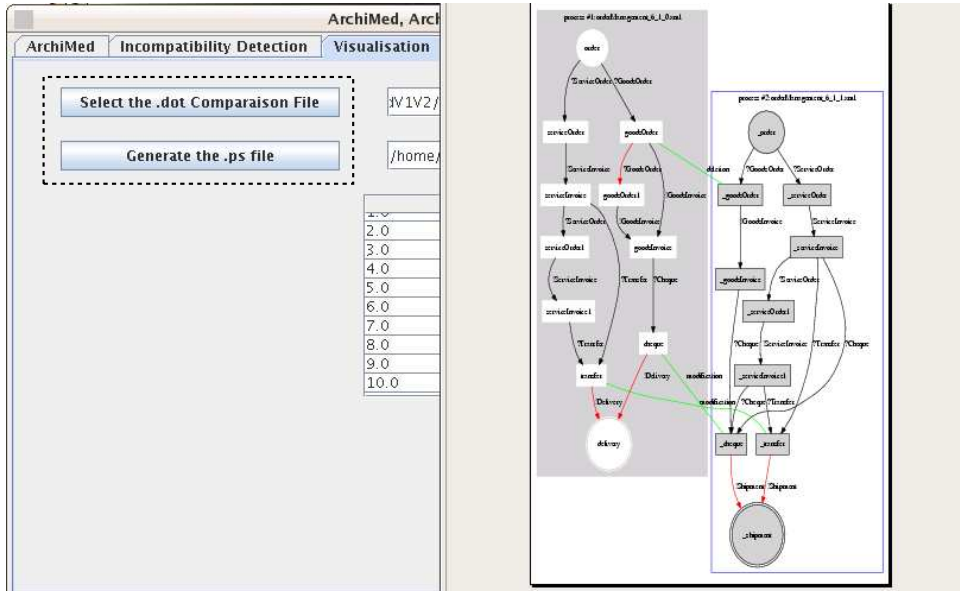


FIG. 6.8 – Résultats sous forme graphique de la détection des incompatibilités

La visualisation comporte deux parties : chacune décrit un graphe qui représente l'un des deux automates comparés. Les paires d'états qui induisent une incompatibilité sont reliées par une arrête qui porte le libellé de l'incompatibilité détectée (*ajout*, *suppression* ou *modification* d'opération).

6.3.3 Mesure de similarité entre les interfaces

L'autre fonctionnalité du canevas permet de détecter les incompatibilités entre une interface et une collection d'interfaces. Ceci s'appuie sur la mesure de similarité que nous avons présentée dans la section 5.4.3 du chapitre 5. Cette mesure de similarité est utilisée pour la sélection du médiateur approprié à l'interface requise d'un client qui sollicite le service selon une des descriptions de son interface fournie. Des tests sur des collections d'interfaces fournies peuvent être menés afin de mesurer la similarité d'une interface par rapport à l'interface inverse de l'interface requise d'un client dans la perspective d'en choisir la plus proche. Le canevas peut alors sélectionner le médiateur sur la base des définitions des interfaces qui engendrent un nombre minimum d'incompatibilités.

La figure 6.9 illustre la comparaison d'une interface `orderManagement-6-1-0.xml` à une collection d'interfaces. Le répertoire de la collection des interfaces est sélectionné en utilisant le bouton `Interface collection : scxml descriptions`.

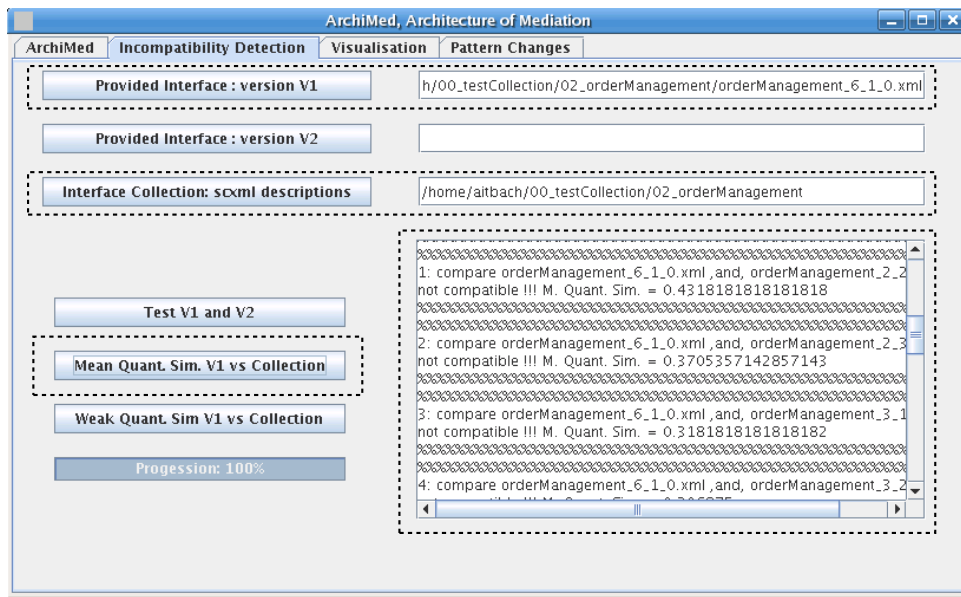


FIG. 6.9 – Résultats textuels de la mesure de similarité

Le bouton *Mean Quant. Sim. V1 vs Collection*¹¹ permet de calculer la mesure de similarité moyenne entre l'interface *V1* et chacune des interfaces de la collection des interfaces. Les résultats textuels sont affichés dans la zone de texte. Pour chaque paire d'interfaces comparées un résultat numérique entre zéro et un leur est associé. Cette valeur est la mesure de similarité entre ces deux interfaces.

Le bouton *Weak Quant. Sim. V1 vs Collection*¹² permet d'appliquer une autre mesure de similarité telle que présentée dans d'autres travaux [100, 74].

6.3.4 Détection des incompatibilités automatiquement résolubles

Un compilateur Java pour Prolog¹³ est intégré au canevas *ArchiMed*. Ce compilateur permet d'évaluer les expressions écrites en Prolog qui modélisent les incompatibilités qui peuvent être résolues automatiquement. Ces expressions sont représentées par des règles qui seront évaluées sur des faits. La base de faits contient les descriptions des deux LTSs à comparer. En d'autres termes, les faits définis par le prédicat *Transition(s1, s2, t)* indiquent toutes les transitions des deux automates où : *s1* est l'état source de la transition *t* et *s2* est l'état cible.

¹¹Mean Quantitative Simulation

¹²Weak Quantitative Simulation

¹³<http://jlogic.sourceforge.net/>

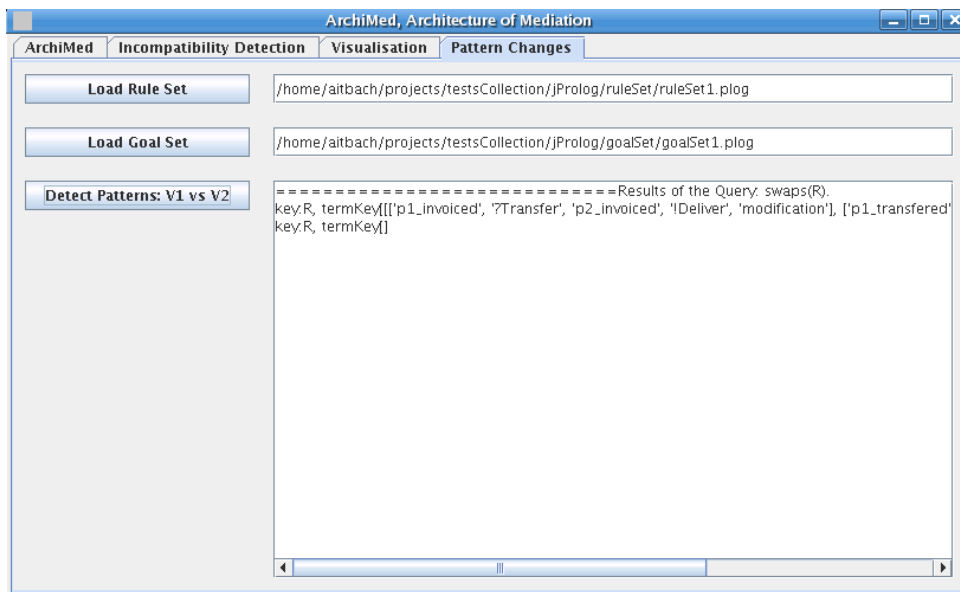


FIG. 6.10 – Détection des incompatibilités résolubles

D'autres faits indiquent les états initiaux et les états finals. Les faits des incompatibilités sont représentés par le prédicat *basic_change*(*s1*, *t1*, *s2*, *t2*, *change*) où :

- *s1* est un état du LTS de la précédente version et *t1* une de ses transitions sortantes.
- *s2* est un état du LTS de la version actuelle et *t2* une de ses transitions sortantes.
- *change* est le type de changement de base (ajout, suppression ou modification).

La figure 6.10 illustre l'utilisation de la partie **Pattern Change** pour l'identification des cas d'incompatibilités résoluble. Le bouton **Load Rule Set** permet de charger l'ensemble des expressions qui représentent les cas d'incompatibilités résoluble. La base des faits qui contient les descriptions des automates et des incompatibilités détectées est construite. Une fois la base de faits et des règles Prolog est chargée, elle peut être interrogée par des requêtes en utilisant le bouton **Load Goal Set**. Le résultat est un sous-ensemble des incompatibilités détectées qui sont automatiquement résolubles décrites dans la zone de texte.

6.4 Tests sur la détection des incompatibilités

Afin de valider le canevas *ArchiMed* de manière expérimentale, une base de tests a été mise au point. Elle est constituée des interfaces qui modélisent des processus de commandes d'articles (voir la section 6.4.1). Les tests de la mesure de similarité des interfaces sont introduits dans la section 6.4.2. Dans la section 6.4.3, les résultats de nos tests sont comparés aux résultats obtenus dans d'autres travaux similaires.

6.4.1 Construction de la base de tests

Pour réaliser les tests sur la détection des incompatibilités nous nous sommes appuyés sur vingt descriptions de processus métiers dans le cadre de la gestion des commandes. Ces descriptions sont données de manière informelle dans des documents conformes au standard d'xCBL (*XML Common Business Library*) [59]. Les descriptions des processus ont été retranscrites manuellement en descriptions d'automates et plus précisément dans le format SCXML. Puis, chaque description de processus décrite en SCXML est comparée aux autres descriptions.

Dans cette base de tests, la comparaison de deux interfaces, qui correspondent à deux processus, peut être interprétée comme un changement dans la définition d'une interface pour obtenir une nouvelle définition de l'interface fournie. Par exemple, dans un processus de gestion de commande, le client peut modifier une commande alors que dans un autre processus il n'a plus cette possibilité. Cette incompatibilité est détectée comme une ou plusieurs suppressions d'opérations.

6.4.2 Tests de similarité des interfaces des services

La figure 6.11 illustre les résultats des comparaisons de l'interface d'un processus de commande avec le reste des processus de notre collection de tests.

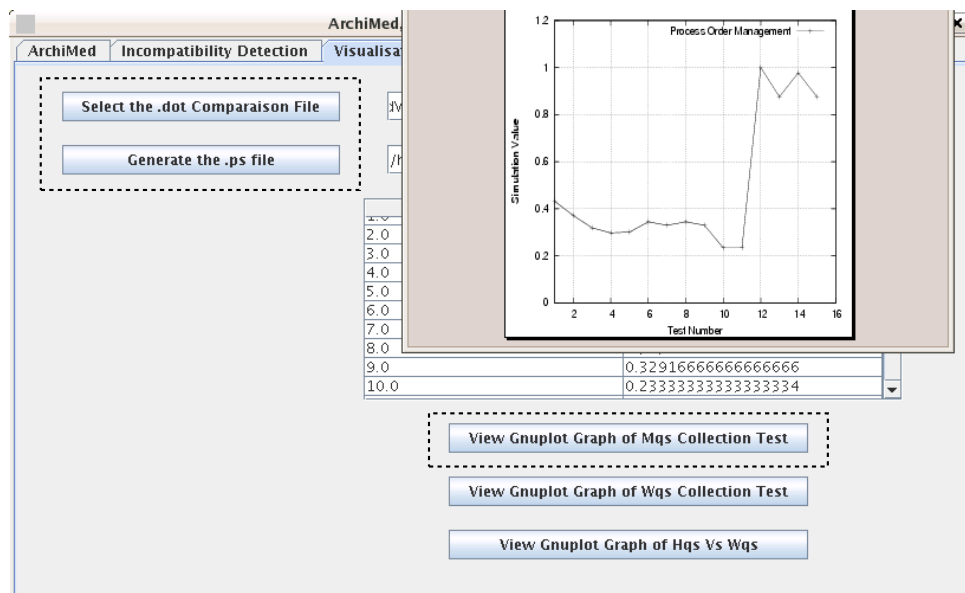


FIG. 6.11 – Résultats graphiques de la mesure de similarité

Les résultats sont présentés sous forme d'une courbe graphique qui indique le numéro de la comparaison et la valeur de la mesure de similarité associée. L'outil offre une repré-

sensation sous forme de tableau des tests et des résultats. La première colonne indique le numéro du test et la deuxième colonne indique la valeur de similarité associé au test. Les interfaces les plus similaires à la requête sont celles qui engendrent le moins d'incompatibilités et dont la mesure de similarité est proche de la valeur un.

6.4.3 Etudes comparatives

Nous avons comparé notre algorithme de détection des incompatibilités ainsi que la mesure de similarité des interfaces à d'autres propositions sur deux plans : le plan théorique et le plan expérimental. Sur le plan théorique, nous explicitons les limites des algorithmes des autres approches et la technique utilisée dans notre algorithme pour lever ces insuffisances. Sur le plan expérimental, nous avons comparé sur une collection de tests notre mesure de similarité des interfaces avec d'autres mesures de similarité proposées par d'autres approches.

Etude théorique

Les algorithmes de détection des incompatibilités qui existent permettent uniquement de vérifier si une interface fournie P est conforme ou non avec une interface requise R . Dans [13], les auteurs proposent des algorithmes de test de conformité entre les interfaces en s'appuyant sur des algorithmes de bisimulation entre l'interface fournie et l'interface contraire de l'interface requise \bar{R} . L'algorithme de bisimulation parcourt les paires d'états (s, s') des deux automates P et \bar{R} de manière synchrone et compare les transitions sortantes des paires d'états atteignables. Dans le cas où pour une paire d'état atteignable il n'existe pas de chemin pour atteindre une paire d'états finals, alors les deux interfaces ne se simulent pas. L'algorithme s'arrête et retourne la valeur faux en résultat.

Dans [117], les auteurs proposent un algorithme de simulation de deux interfaces représentées par des automates A_1 et A_2 en vérifiant l'atteignabilité de l'intersection des deux automates. L'intersection $A_1 \cap A_2$ est un automate dont l'état initial est la paire d'états initiaux (s_{10}, s_{20}) de A_1 et A_2 respectivement. Pour vérifier si les deux automates se simulent, il s'agit de vérifier si à partir de l'état initial (s_{10}, s_{20}) tous les états finals sont atteignables. Lorsqu'un à partir d'un état courant il n'existe pas de chemin vers un état final alors les deux automates ne se simulent pas. L'algorithme s'arrête et retourne la valeur faux.

Dans notre algorithme de détection, lorsqu'une incompatibilité est détectée le parcours ne s'arrête pas. En effet, pour chaque incompatibilité détectée un parcours spécifique y est associée afin de déterminer les paires d'états à visiter. De manière générale, il s'agit de

d'avancer dans la transition à l'origine de l'incompatibilité vers son état cible afin détecter les autres incompatibilités dans l'automate initialié à cet état cible.

Dans [100, 74], les auteurs proposent des algorithmes de bisimulation pour détecter les incompatibilités entre les interfaces. Les algorithmes proposés s'appliquent sur des automates acycliques. Lors du parcours synchrone des deux automates A_1 et A_2 pour déterminer si A_2 simule le comportement de A_1 , l'algorithme vérifie toutes les combinaisons des transitions sortantes d'un état s_1 de A_1 et celles d'un état s_2 de A_2 . Les auteurs définissent une fonction dite *Weak Quantitative Simulation* notée WQS. La fonction WQS : $S \times S \rightarrow [0, 1]$, est définie pour toute paire d'états (s_1, s_2) qui vérifie la condition ci-après :

$$WQS(s_1, s_2) = \begin{cases} 1 & \text{si } s_1 \in F_1, \text{ état final de } A_1 \\ (1 - p) + \max(W_1(s_1, s_2), W_2(s_1, s_2)) & \text{sinon} \end{cases} \quad (6.1)$$

où :

$$W_1(s_1, s_2) = \max_{t_2 \in s_2 \bullet} (L(\varepsilon, l_2) \cdot WQS(s_1, t_2 \circ)) \quad (6.2)$$

$$W_2(s_1, s_2) = \frac{p}{n} \sum_{t_1 \in s_1 \bullet} \max \left(L(l_1, \varepsilon) \cdot WQS(t_1 \circ, s_2), \max_{t_2 \in s_2 \bullet} (L(l_1, l_2) \cdot WQS(t_1 \circ, t_2 \circ)) \right) \quad (6.3)$$

$$n = \|s_1 \bullet\|, l_1 = \text{Label}(t_1), l_2 = \text{Label}(t_2), p \in [0, 1]$$

L'algorithme est défini comme une fonction de maximisation de la valeur de simulation d'une paire d'états (voir l'expression 6.1). Chaque transition sortante t de s_1 ou de s_2 est supposée être une transition dont le libellé est celui d'une opération ajoutée (voir l'expression 6.2), supprimée ou modifiée (voir l'expression 6.3).

Une pénalité est associée à chaque type d'incompatibilité. A cette effet, les auteurs ont défini la fonction $L : I \cup \{\varepsilon\} \times I \cup \{\varepsilon\} \rightarrow [0, 1]$, où I est l'ensemble des libellés d'un automate. De plus, un facteur p est introduit et donne le poids de la mesure de similarité pour une paire d'états ainsi que pour les états descendants. La fonction de maximisation permet de déterminer laquelle des combinaisons des cas d'incompatibilité retourne la plus grande valeur de simulation pour une paire d'états. Cette fonction de maximisation est appliquée récursivement sur les états cibles des transitions comparées.

Dans cette approche, les auteurs n'ont pas proposé une modélisation précise de cas d'incompatibilité. En effet, le résultat des algorithmes ne retourne que la valeur de simulation entre les deux automates et ne permet pas d'identifier avec précision la localisation des incompatibilités sur les deux automates. Cette approche a également la particularité

d'être basée sur une exploration exhaustive de l'espace des combinaisons des cas d'incompatibilité dans les deux automates. Cette exploration a un coût où la complexité des algorithmes est exponentielle voir factorielle d'après leurs auteurs. Dans notre approche, nous avons, au contraire, identifier les conditions qui caractérisent chaque incompatibilité en utilisant entre autres des mécanismes d'exploration en aval des automates. Dans notre algorithme de détection, nous avons construit des ensembles des transitions dont les libellés sont équivalents afin de ne pas les considérer comme des incompatibilités, réduisant ainsi l'espace d'exploration des deux automates comparés. La complexité de l'algorithme ainsi obtenu n'est que quadratique au nombre des transitions des deux automates.

Etude expérimentale des mesures de similarité

Nous avons mené une étude quantitative comparative de notre mesure de similarité moyenne à une autre mesure de similarité (dite *Weak Quantitative Simulation*) décrite dans les travaux de *Sokolsky* [100] et repris par *Lohmann* [74]. Cette mesure est présentée comme une maximisation de la valeur de simulation lors du parcours des automates des deux interfaces comparées. Ces travaux considèrent des graphes acycliques. Ainsi, nous avons dû modifier la base de processus tests pour supprimer les cycles qu'ils contenaient. D'autre part, la mesure de similarité est calculée en associant à chaque cas d'incompatibilité (ajout, suppression, modification d'opérations) la même pénalité de 0,5. Dans l'expérience que nous avons menée, nous comparons chaque interface de la base de tests, à l'interface de référence, en fournissant la mesure proposée dans ces travaux (*Weak Quantitative Simulation*) et la mesure que nous proposons (*Mesure de similarité moyenne*).

Le résultat des évaluations est retourné sous forme de graphes qui indique les courbes associées à chaque mesure de similarité. La figure 6.12 illustre les résultats des comparaisons des deux mesures de similarité des interfaces.

Les résultats obtenus par la mesure WQS (*Weak Quantitative Simulation*) sont des valeurs toutes très proches de un (voir la courbe en trait plein de la figure 6.12). La valeur un désigne qu'une interface simule l'interface donnée en référence. Ces résultats ne sont pas discriminants et ne permettent pas d'identifier les interfaces qui simulent de celles qui ne simulent pas l'interface référence. Cependant, les résultats obtenus par la mesure de similarité moyenne sont discriminants (voir la courbe en pointillés de la figure 6.12).

L'interface référence, contient 11 états et 13 transitions. Le tableau 6.1, illustre le détail de quatre résultats limités aux tests numéro onze à quatorze.

L'interface du test numéro onze contient dix états et quatorze transitions (voir la ligne Num. 11 du tableau 6.1). Les incompatibilités détectées entre cette interface et l'interface référence sont au nombre de dix-neuf. Dans le test numéro 11, la mesure de

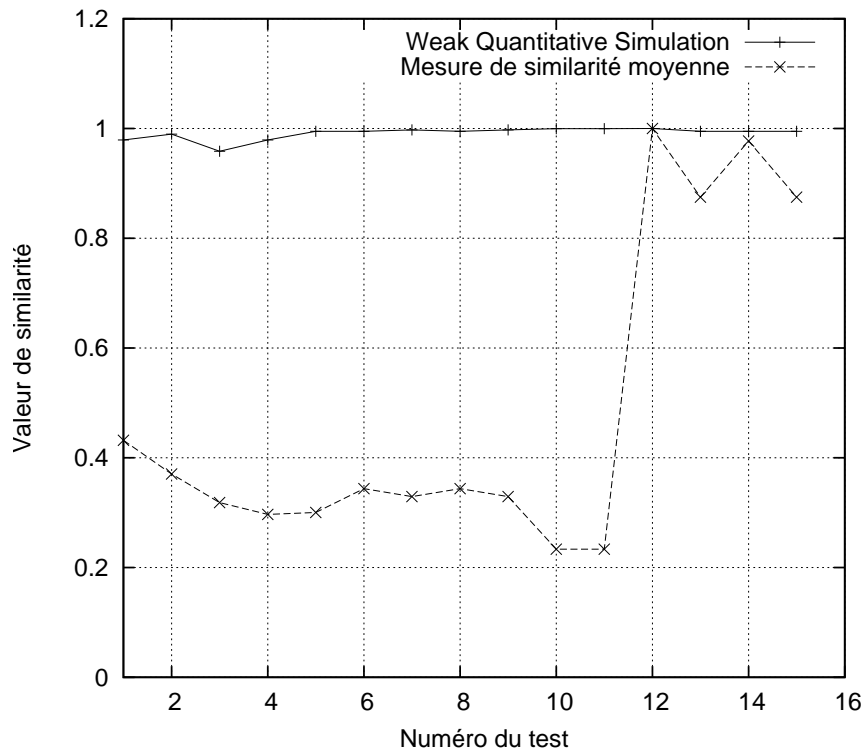


FIG. 6.12 – Résultats des comparaisons des mesures de similarité

similarité moyenne que nous proposons retourne une valeur de 0.233 qui est proche de zéro (voir la colonne Sim. Moy. du tableau 6.1). Cette valeur indique que l'interface du test 11 ne simule pas l'interface référence et les différences entre les deux interfaces sont importantes. Ce résultat est cohérent avec le nombre d'incompatibilités détectées dans ce test. En effet, plus il y a des incompatibilités et moins la mesure de similarité sera proche de un. Cependant, la mesure *Weak Quantitative Simulation* retourne une valeur 0.999 qui est proche de un (voir la colonne WQS du tableau 6.1). Cette valeur indique que l'interface du test 11 ne simule pas l'interface posée comme requête et les différences entre les deux interfaces ne sont pas importantes. Ce résultat n'est pas cohérent avec le nombre d'incompatibilités détectées dans ce test (19 incompatibilités).

Interfaces	Sim. moy.	WQS	Etats	Transitions	Incompatibilités
Num. 11	0.233	0.999	10	14	19
Num. 12	1	1	11	13	0
Num. 13	0.875	0.994	10	13	3
Num. 14	0.977	0.994	11	13	1

TAB. 6.1 – Détails de quelques résultats des tests

L'interface du test numéro douze simule le comportement de l'interface posée comme requête et dans les deux mesures la valeur est un.

Dans les tests treize et quatorze, les valeurs des mesures de similarité moyenne sont différentes (0.875 pour un nombre d'incompatibilités de trois et 0.977 pour une incompatibilité). Cependant, la mesure *Weak Quantitative Simulation* retourne la même valeur 0.994 pour les tests treize et quatorze.

Sur le plan des performances des tests le temps moyen d'exécution de toutes les comparaisons avec la mesure de similarité moyenne est de $150 * 10^{-3}$ seconds. Cependant, le temps moyen d'exécution de toutes les comparaisons avec la mesure *Weak Quantitative Simulation* est plus grand ($13707 * 10^{-3}$ seconds). Cette différence est expliquée par la complexité des algorithmes proposés par les auteurs qui est exponentielle.

6.5 Conclusion

Dans ce chapitre nous avons présenté les aspects techniques de la mise en œuvre du canevas *ArchiMed*. La détection des incompatibilités est réalisée dans un module qui offre, une visualisation graphique des incompatibilités retournées afin de guider les concepteurs des services dans la résolution manuelle des incompatibilités. Le compilateur des expressions assure l'évaluation des expressions de détection des cas d'incompatibilité automatiquement résolubles. Le prototype a été testé sur une base de tests montrant ainsi les différentes fonctionnalités qu'il permet de réaliser. Une étude quantitative et comparative à des travaux similaires est réalisée. Par rapport aux tests effectués, les résultats obtenus montrent que la mesure de similarité que nous proposons est plus discriminante.

La mise en œuvre du canevas a nécessité la définition de 35 classes et le nombre de lignes de code produites est 15303. Une démonstration du canevas est accessible en ligne à l'adresse : <http://www-clips.imag.fr/mrim/User/ali.ait-bachir/webServices/webServices.html>

Dans la mise en œuvre du canevas *ArchiMed*, l'effort a porté sur la détection des incompatibilités. La mise en œuvre de la résolution est en cours.

Chapitre 7

Conclusion générale et perspectives

7.1 Contributions de la thèse

Le canevas ArchiMed est la solution que nous proposons pour la détection des incompatibilités entre deux interfaces de services. Les incompatibilités étudiées sont celles dues à des différences entre les dimensions comportementales des interfaces des services. Notre proposition vient en complément des travaux qui traitent des adaptations structurelles et non-fonctionnelles des interfaces des services [9, 29].

Dans notre approche la conformité entre un service et ses clients est vérifiée aussi bien à la modification de l'interface fournie par le service qu'au moment où un client engage avec lui une conversation.

Lorsque l'interface fournie du service est modifiée, une nouvelle version est générée puis est rendue disponible. Dans le cas où la nouvelle version de l'interface ne simule pas la précédente, toutes les incompatibilités entre les deux versions sont détectées. En fonction de la nature des incompatibilités détectées, il est possible de générer un médiateur qui assure l'adaptation des interactions engagées par un client dont l'interface requise était compatible avec la précédente version de l'interface fournie, mais ne l'est plus avec la nouvelle.

Au moment où un client engage une conversation avec le service, si l'interface qu'il requiert n'est pas conforme avec la version actuelle de l'interface fournie, les interactions sont conduites selon une version antérieure de l'interface fournie, s'il en existe une. La conversation entre le client et service est effectuée via le médiateur préalablement généré au moment de la définition de la version actuelle de l'interface fournie.

Dans d'autres travaux, les versions successives de l'interface du services sont toutes exposées, et les clients continuent d'interagir selon la version qui correspond à l'interface qu'ils requièrent. Aucun mécanisme de détection des incompatibilités entre les interfaces mises en jeu n'est proposé [109, 71].

Dans la détection automatique des incompatibilités, la plupart des approches proposent des mécanismes qui comparent des interfaces et qui indiquent uniquement si les deux interfaces comparées sont compatibles ou non [13, 117]. D'autres approches proposent des mesures de similarité (synonyme d'un certain degré de compatibilité) entre les interfaces, sans pour autant détecter avec exactitude sur les interfaces comparées, les endroits où les incompatibilités se sont produites [26, 74]. Une originalité de notre proposition est la localisation avec précision dans les interfaces, des emplacements des différentes incompatibilités afin de les résoudre [3].

Les interfaces étant modélisées par des automates déterministes en nombre fini d'états (LTS), les incompatibilités due à l'ajout, la suppression et la modification d'opérations sont formalisées par le biais d'expressions booléennes sur les automates. Nous avons démontré que ces expressions s'excluent mutuellement. Une étude de leur complétude a démontré que tous les cas d'ajout, de suppression et de modification d'opérations sont pris en compte. Ces expressions sont utilisées dans l'algorithme que nous proposons pour la détection. Ce dernier parcourt les LTSs des interfaces comparées et ne s'arrête pas à la première incompatibilité trouvée. En effet, l'algorithme continue la détection jusqu'à trouver toutes les incompatibilités qui existent entre les deux interfaces. Ainsi, l'algorithme retourne les localisations exactes de toutes les incompatibilités, afin qu'elle soient visualisées graphiquement, si nécessaire.

Pour résoudre les incompatibilités certains travaux proposent de propager les changements dans l'interface du service aux applications clientes qui l'utilisent [17, 95]. Toutefois, cette propagation n'est pas applicable dans le contexte du web, car les applications clientes sont indépendantes et sous le contrôle d'une autre organisation. La résolution automatique des incompatibilités, lorsqu'elle est possible, assure la transparence des évolutions des interfaces fournies pour les clients. Dans notre approche, nous utilisons un mécanisme de sélection d'un médiateur pour la résolution automatique des incompatibilités entre le service et un client selon son interface requise [2]. A cet effet, nous avons proposé une mesure de similarité des interfaces pour découvrir le médiateur approprié à la conversation initié selon l'interface requise du client. Notre mesure de similarité des interfaces a apporté des améliorations significatives comparées à d'autres mesures présentées dans des travaux similaires [100, 74].

7.2 Limites et perspectives

Dans la détection des incompatibilités nous avons identifié des incompatibilités élémentaires d'ajout, de suppression et de modification d'opérations. Ces incompatibilités peuvent être regroupées pour désigner des incompatibilités composées [114]. Une des pers-

pectives de la thèse et d'étudier les expressions qui modélisent ces incompatibilités pour les détecter et les résoudre.

Dans la résolution des incompatibilités nous n'avons présenté que deux cas qui peuvent être résolus automatiquement : i) la suppression d'un terme dans une alternative et ii) la suppression d'une opération d'acquiescement dans une séquence. Une des perspectives est d'étudier les autres incompatibilités qui peuvent être détectées et résolues automatiquement. Certaines sont décrites dans [29].

Certains cas où la résolution ne peut pas être automatisée correspondent aux situations où le médiateur ne peut pas avoir une information du client ou du service si ces derniers ne la lui envoient pas. Une résolution semi-automatique peut être envisagée du côté des applications clientes. Il s'agit d'offrir à un concepteur de résoudre ces incompatibilités directement sur le graphe retourné en résultats. Une idée est d'utiliser les outils offerts par Eclipse tel que EMF qui permet des transformations de modèles sur la base de leurs méta-modèle. Le recours aux communautés de services peut être envisagé [28]. Le service fournisseur qui a évolué sera ainsi substitué par un service qui reste compatible avec l'interface requise du client.

La mise en œuvre de la résolution des incompatibilités peut être complétée par des mécanismes de transformation des représentations des interfaces en automates vers des représentations en BPEL. Il s'agit de déployer les médiateurs en utilisant les moteurs d'exécution de processus qui existent (voir l'annexe D).

Dans le cas où les incompatibilités ne peuvent être résolues par la génération automatique de médiateurs, le fournisseur inscrit à un registre de services peut proposer à ces clients de rechercher des partenaires dont l'interface fournie est équivalente à sa précédente version. Cette recherche peut évidemment être envisagée du côté du client mais l'idée est de maintenir la compatibilité des interfaces requise par les clients avec les services. Le médiateur, dans ce cas, joue le rôle d'aiguilleur vers d'autres services. Dans le cas où le service localisé par le médiateur évolue, le médiateur (aiguilleur) va sélectionner (par la découverte de service) un nouveau service qui est équivalent à l'interface précédente exigée par ce même client. Il s'agit de s'appuyer sur la mesure de similarité entre les interfaces des services que nous avons proposée.

Annexe A

Démonstrations de l'exclusion mutuelle

Démonstration de $Sup \oplus Mod$:

D'après les expressions de détection de suppression et de modification d'opérations explicitées dans la section 4.2.2 et dans la section 4.2.4, nous avons $Sup = Eq4.1 \vee Eq4.2$ et $Mod = Eq4.5 \wedge Eq4.6$. Par conséquent, pour démontrer que $Sup \oplus Mod$ il faut démontrer que : $(Eq4.1 \oplus (Eq4.5 \wedge Eq4.6)) \wedge (Eq4.2 \oplus (Eq4.5 \wedge Eq4.6))$.

5) *Preuve de $Eq4.1 \oplus (Eq4.5 \wedge Eq4.6)$:*

Supposons que $Eq4.1$ est vraie pour une paire d'états (s, s') . Supposons, également, que $Eq4.5 \wedge Eq4.6$ est vérifiée pour la même paire d'états (s, s') . Soit t' une transition sortante de s' dont le libellé $Label(t')$ est celui d'une opération modifiée par une autre opération. D'après $Eq4.5$ nous avons $Label(t') \notin Label(s\bullet)$. Or, d'après $Eq4.1$ nous avons $\|Label(s'\bullet) - Label(s\bullet)\| = 0$. Donc, le résultat $Eq4.1 \oplus (Eq4.5 \wedge Eq4.6)$ est vérifié.

6) *Preuve de $Eq4.2 \oplus (Eq4.5 \wedge Eq4.6)$:*

Supposons que $Eq4.2$ est vraie pour une paire d'états (s, s') . Supposons, également, que $Eq4.5 \wedge Eq4.6$ est vérifiée pour la même paire d'états (s, s') . Soit t une transition sortante de s dont le libellé $Label(t)$ est celui d'une opération supprimée. On suppose que cette même transition t est une transition dont le libellé $Label(t)$ à été modifié par une autre opération. D'après $Eq4.2$ nous avons que $\exists t' \in s'\bullet : ExtIn(t', (t\circ)\bullet)$. Or d'après $Eq4.6$ nous avons la négation de l'expression précédente qui est $\neg \exists t' \in s'\bullet : ExtIn(t', (t\circ)\bullet)$. De ce fait $Eq4.2 \oplus (Eq4.5 \wedge Eq4.6)$ est vraie.

Conclusion de la démonstration de $Sup \oplus Mod$:

D'après les preuves (5) et (6) nous avons $(Eq4.1 \oplus (Eq4.5 \wedge Eq4.6)) \wedge (Eq4.2 \oplus (Eq4.5 \wedge Eq4.6)) \Rightarrow Sup \oplus Mod$. Les expressions de détection de suppression et de modification d'opérations s'excluent mutuellement.

Démonstration de $Aj \oplus Mod$:

D'après les expressions de détection d'ajout et de modification d'opérations explicitées dans la section 4.2.3 et dans la section 4.2.4, nous avons $Aj = Eq4.3 \vee Eq4.4$ et $Mod = Eq4.5 \wedge Eq4.6$. Par conséquent, pour démontrer que $Aj \oplus Mod$ il faut démontrer que $:(Eq4.3 \oplus (Eq4.5 \wedge Eq4.6)) \wedge (Eq4.4 \oplus (Eq4.5 \wedge Eq4.6))$.

7) Preuve de $Eq4.3 \oplus (Eq4.5 \wedge Eq4.6)$:

Supposons que $Eq4.3$ est vraie pour une paire d'états (s, s') . Supposons, également, que $Eq4.5 \wedge Eq4.6$ est vérifiée pour la même paire d'états (s, s') . Soit t une transition sortante de s dont le libellé $Label(t')$ est celui d'une opération modifiée par une autre opération. D'après $Eq4.5$ nous avons $Label(t) \notin Label(s' \bullet)$. Or, d'après $Eq4.3$ nous avons $\|Label(s \bullet) - Label(s' \bullet)\| = 0$. Donc, le résultat $Eq4.3 \oplus (Eq4.5 \wedge Eq4.6)$ est vérifié.

8) Preuve de $Eq4.4 \oplus (Eq4.5 \wedge Eq4.6)$:

Supposons que $Eq4.4$ est vraie pour une paire d'états (s, s') . Supposons, aussi, que $Eq4.5 \wedge Eq4.6$ est vérifiée pour la même paire d'états (s, s') . Soit t' une transition sortante de s' dont le libellé $Label(t')$ est celui d'une opération supprimée. On suppose que cette même transition t' est une transition dont le libellé $Label(t')$ à été modifié par une autre opération. D'après $Eq4.4$ nous avons que $\exists t \in s \bullet : ExtIn(t, (t' \circ) \bullet)$. Or d'après $Eq4.6$ nous avons la négation de l'expression précédente qui est $\neg \exists t \in s \bullet : ExtIn(t, (t' \circ) \bullet)$. De ce fait $Eq4.4 \oplus (Eq4.5 \wedge Eq4.6)$ est vraie.

Conclusion de la démonstration de $Sup \oplus Mod$:

D'après les preuves (7) et (8) nous avons : $(Eq4.3 \oplus (Eq4.5 \wedge Eq4.6)) \wedge (Eq4.4 \oplus (Eq4.5 \wedge Eq4.6)) \Rightarrow Aj \oplus Mod$. Les expressions de détection d'ajout et de modification d'opérations s'excluent mutuellement.

Annexe B

Exemple description en WSDL d'un service web

La description de l'interface d'un service en WSDL contient plusieurs parties qui sont :

- *types* : décrit des type de donnée structurées à partir d'autres types de base (entier, réel, caractère, chaîne, etc.) et/ou d'autres types structurés,
- *message* : décrit le nom du message ainsi que les différentes parties qui le constituent,
- *portType* : définit les signatures des opérations, les paramètres des opérations et leurs types,
- *binding* : établit le mécanisme de communication du service web

Les éléments de description d'une interface de service Web en WSDL sont données ci-après :

```

<definitions>
  <types>
    definition des types
  </types>
  <message>
    definition d'un message
  </message>
  <portType>
    definition des opérations du service
  </portType>
  <binding>
    definition du protocole des interactions
  </binding>
</definitions>

```

Soit par exemple, le service *entrepôt* (*warehouseItems*) un service web qui offre une opération de consultation du libellé complé d'un article à partir de son identifiant. Le fichier WSDL qui décrit l'interface du service *entrepôt* est donné comme suit :

```
<definitions>

  <message name="getByIdItemRequest">
    <part name="IdItem" type="xs:string"/>
  </message>

  <message name="getByIdItemResponse">
    <part name="value" type="xs:string"/>
  </message>

  <portType name="warehouseItems">
    <operation name="getItem">
      <input message="getByIdItemRequest"/>
      <output message="getByIdItemResponse"/>
    </operation>
  </portType>

  <binding type="warehouseItems" name="b1">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <operation>
      <soap:operation
        soapAction="http://example.com/getItem"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
</definitions>
```

Dans cet exemple, les messages décrits sont des messages en entrée et en sortie de type chaîne de caractères. L'opération de consultation des libellés des articles est `getItem` qui possède deux paramètres : un premier paramètre en entrée et un deuxième paramètre en sortie. Les messages échangés sont dans le format SOAP et le protocole de communication est *HTTP*.

Annexe C

Intervalle de définition de la mesure de similarité moyenne

$$W_{qs} \in [0, 1]$$

Le score de similarité W_{qs} d'une paire d'états visitée (si, sj) dans l'algorithme de détection est compris entre zéro et un. La démonstarion de ce résultat est donné comme suit :

- 1 $W_{qs}(si, sj) = 1$ { lorsque si n'a pas de transitions sortantes }
- Losque si a des transitions sortantes :
- 2 posons $\| Diff((si, sj)) \| = n$ { $n \in \mathbb{N}^+$ }
- 3 posons $\| Label(si\bullet) \cap Label(sj\bullet) \| = m$ { $m \in \mathbb{N}^+$ }
- 4 $n + m \neq 0$
- 5 $0 \leq Weight(D_1) < 1$
- 6 $0 \leq Weight(D_2) < 1$
- 7 ...
- 8 $0 \leq Weight(D_n) < 1$
- 9 $0 \leq \sum_{i=1}^n Weight(D_i) < n$
- 10 $m \leq \sum_{i=1}^n Weight(D_i) + m < n + m$
- 11 $\frac{m}{n+m} \leq \frac{\sum_{i=1}^n Weight(D_i) + m}{n+m} < 1$
- 12 $0 \leq \frac{\sum_{i=1}^n Weight(D_i)}{n} < 1$ { pour $m = 0, (n \neq 0)$ }
- 13 $W_{qs}(si, sj) = 0$ { Lorsque les valeurs des pénalités sont toutes égales à zéro }

$$Mqs \in [0, 1]$$

La mesure de similarité moyenne Mqs entre deux LTSs P_i et P_j est comprise entre zéro et un. La démonstarion de ce résultat est donné comme suit :

- 1 posons $\| \text{visitées} \| = n$ $\{ n \in \mathbb{N}^{*+} \}$
- 2 $0 \leq Mqs(PS_1) \leq 1$ $\{ \text{D'après la démonstaration ci-avant} \}$
- 3 $0 \leq Mqs(PS_2) \leq 1$
- 4 ...
- 5 $0 \leq Mqs(PS_n) \leq 1$
- 6 $0 \leq \sum_{i=1}^n Wqs(PS_i) \leq n$
- 7 $0 \leq \frac{\sum_{i=1}^n Wqs(PS_i)}{n} \leq 1$ $\{ n \neq 0 \}$

Annexe D

Transformation de modèles SCXML vs BPEL

Dans la phase de génération des médiateurs, le LTS produit doit être interprété par un moteur d'exécution afin de générer des instances de processus à chaque communication initiée par un client. La représentation en SCXML des LTSs peut être exécutée par un moteur dit CommonSCXML du groupe Apache. Cependant, cette solution implique que le standard SCXML doit être accepté comme un nouveau standard pour la représentation et l'exécution des processus métiers dans les services web. Cela implique une définition de nouveaux mécanismes d'interaction entre les services. Ces mécanismes existent déjà dans le standard BPEL où de nombreux moteurs d'exécution ont été développés (par exemple, WebSphere, BPELeclipse, etc.). De ce fait, nous avons opté pour une utilisation de moteurs d'exécution de BPEL et pour ce faire nous devons développer un outil de transformation de modèle de SCXML vers BPEL.

Nous avons étudié quelques outils qui existe sur la transformation de modèle mais aucun ne traitent de transformation des automates modélisés en SCXML vers BPEL. L'outil *WS-Engineer*¹⁴ est un plugin Eclipse qui permet d'effectuer une transformation de modèles en BPEL vers des représentation en LTS mais pas l'inverse. L'outil *Petrify*¹⁵ s'appuie sur des représentations en réseaux de Pétri pour effectuer des tests de terminaison et permet des réductions de graphes en éliminant les opérations internes entre autres. Il génère une représentation en automate mais ne propose pas de transformation en BPEL.

Dans [81], les auteurs proposent des algorithmes qui permettent de transformer des structures de programmes avec des *GoTo* vers des formes plus structurées en *Si-alors-Sinon* en boucle *Tant que*. Cette transformation peut être utile dans la transformation des automates en BPEL.

¹⁴<http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/>

¹⁵<http://www.lsi.upc.edu/jordicf/petrify/home.html>

Bibliographie

- [1] M. Aiello, M. P. Papazoglou, J. Yang, M. Carman, M. Pistore and L. Serafini, and P. Traverso. A request language for web-services based on planning and constraint satisfaction. In *Proceedings of the Third International Workshop on Technologies for E-Services*, volume 2444 of *LNCS*, pages 76–85, Hong Kong, China, August 2002. Springer-Verlag. pages 43, 65
- [2] A. Ait-Bachir, M. Dumas, and M.-C. Fauvet. BESERIAL : Behavioural service analyser. In *Proceedings of the International Conference on Business Process Management, demo session*, number 5240 in *LNCS*, pages 374–377, Milano, Italy, Sptember 2008. Springer Verlag. pages 23, 123, 136
- [3] A. Ait-Bachir and M.-C. Fauvet. ArchiMed : canevas multi-médiateur pour la réconciliation de conversations entre services web. *Ingénierie des Systèmes d’Information, Lavoisier*, 13(4) :83–106, 2008. pages 23, 136
- [4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer-Verlag, Berlin, 2003. pages 8, 30
- [5] M. Altenhofen, E. Boerger, and J. Lemcke. An execution semantics for mediation patterns. In *Proceedings of the BPM’2005, Workshops : Workshop on Choreography and Orchestration for Business Process Managament*, Nancy, France, September 2005. pages 29, 58, 61, 65, 66, 78
- [6] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara. DAML-S : Web service description for the semantic web. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 348 – 363, Sardinia, Italy, June 2002. pages 41, 64, 65
- [7] L. Ardissono, A. Goy, and G. Petron. Enabling conversations with web services. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 819–826, Melbourne, Victoria, Australia, July 2003. ACM. pages 30

- [8] A. Awad and F. Puhlmann. Structural detection of deadlocks in business process models. In *Proceedings of the International Conference on Business Information Systems*, number 7 in Lecture Notes in Business Information Processing, pages 239–250, Innsbruck, Austria, May 2008. Springer. pages 32
- [9] 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, June 2005. Springer Verlag. pages 27, 28, 58, 64, 65, 66, 70, 135
- [10] B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv environment for web services composition. *Internet Computing, IEEE*, 7(1) :40–48, 2003. pages 12, 40
- [11] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proceedings of the 14th international conference on World Wide Web*, pages 148 – 159, Keio, Japan, May 2005. ACM, New York, USA. pages 70
- [12] L. Bordeaux and G. Salaün. Using process algebra for web services : Early results and perspectives. In *Proceedings of the 5th International Workshop in Technologies for E-Services (TES)*, number 3324 in LNCS, pages 54–68, Toronto, Canada, August 2004. Springer Verlag. pages 31, 65
- [13] 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*, number 3324 in LNCS, pages 15–28, Toronto, Canada, August 2004. Springer Verlag. pages 14, 16, 31, 44, 64, 65, 70, 76, 130, 136
- [14] M. Brambilla, G. Guglielmetti, and C. Tziviskou. Asynchronous web services communication patterns in business protocols. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, number 3806 in Lecture Notes in Computer Science, pages 435–442, New York, USA, November 2005. Springer Verlag. pages 30
- [15] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification : A new approach to design and analysis of E-Service composition. In *Proceedings of the International Conference on World Wide Web*, pages 403 – 410, Budapest, Hungary, May 2003. ACM. pages 31, 35
- [16] J. Cardoso¹, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Journal of Web Semantics, Elsevier*, 1(3) :281–308, April 2004. pages 10, 40
- [17] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data and Knowledge Engineering, Springer Verlag*, 24 :211–238, 1998. pages 49, 63, 64, 136

-
- [18] P.-A. Champin and C. Solnon. Measuring the similarity of labeled graphs. In *Proceedings of the 5th International Conference. On Case-Based Reasoning (ICCBR)*, number 2689 in Lecture Notes in Computer Science, pages 80–95, Trondheim, Norway, June 2003. Springer. pages 37, 59
- [19] P. Chrzastowski-Wachtel, B. Benatallah, R. Hamadi, M. O’Dell, and A. Susanto. A top-down petri net-based approach for dynamic workflow modeling. In *Proceedings of the International Conference on Business Process Management*, number 2678 in LNCS, pages 336–353, Eindhoven, The Netherlands, June 2003. Springer Verlag. pages 31, 32, 51
- [20] E.-M. Clarke, O. Grumberg, and D.-A. Peled. *Model Checking*, chapter Equivalences and Preorders between Structures, pages 171–178. The MIT Press, Cambridge (Mass.), London, 2001. pages 16, 70, 76
- [21] J. Colgrave, R. Akkiraju, and R. Goodwin. External matching in UDDI. In *Proceedings of the IEEE International Conference on Web Services*, pages 226–233, San Diego, USA, July 2004. IEEE. pages 7
- [22] J. H. Conway. *Regular Algebra and Finite Machines*. William Clowes and Sons Ltd, university college of north wales, bangor and balliol college, oxford edition, 1971. pages 38
- [23] J. C. Corrales, D. Grigori, and M. Bouzeghoub. BPEL processes matchmaking for service discovery. In *Proceedings of the OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE*, number 4275 in LNCS, pages 237–254, Montpellier France, October-November 2006. Springer. pages 35, 39, 58
- [24] K. H. Dam. An agent-oriented approach to support change propagation in software evolution. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1736–1737, Estoril, Portugal, May 2008. ACM. pages 17
- [25] P. Derler and R. Weinreich. Models and tools for SOA governance. In *Proceedings of the International Conference on Trends in Enterprise Application Architecture (TEAA)*, volume 4473 of LNCS, pages 112–126, Berlin, Germany, November, December 2006. Springer. pages 6, 57
- [26] R. Dijkman. Diagnosing differences between business process models. In *Proceedings of the International Conference on Business Process Models*, number 5240 in LNCS, pages 261–277, Milano, Italy, September 2008. Springer. pages 136
- [27] R. Dijkman and M. Dumas. Service-oriented design : A multi-viewpoint approach. *International Journal of Cooperative Information Systems, World Scientific Publishing Company*, 13(4) :337–368, 2004. pages 32, 33, 83

- [28] H. Duarte. TCOWS : canevas pour la composition de services web avec propriétés transactionnelles. Thèse de Doctorat de l'Université Joseph Fourier, 2007. pages 40, 137
- [29] M. Dumas, M. Spork, and K. Wang. Adapt or perish : Algebra and visual notation for service interface adaptation. In *Proceedings of the 4th International Conference on Business Process Management (BPM)*, number 4102 in LNCS, pages 65–80, Vienna, Austria, September 2006. Springer Verlag. pages 7, 15, 22, 27, 28, 58, 60, 65, 66, 135, 137
- [30] C. Ellis, K. Keddera, and G. Rozenberg. Dynamic change within workflow systems. In *Proc. of conference on Organizational computing systems COCS '95*, pages 10–21, Milpitas, California, United States, August 1995. ACM Press. pages 50, 63
- [31] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. pages 6
- [32] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 152–161, Montreal, Canada, October 2003. IEEE Computer Society Press. pages 31
- [33] H. Foster, S. Uchitel, J. Magee, and J. Kramer. WS-Engineer : A tool for model-based verification of web service compositions and choreography. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 771–774, Shanghai, China, May 2006. pages 70
- [34] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *International Journal of Web Services (JWSR)*, Idea Group Publishing, 2(4) :68–93, 2005. pages 31, 35
- [35] D. Giribet. Merging XPath and URLs for enhanced web and web service data retrievals. In *Proceedings of the IADIS International Conference on Applied Computing*, volume 2, pages 27–33, Algarve, Portugal, February 2005. pages 27
- [36] Z. Guo, X. Wang, and A. Zhou. Wsquery : Xquery for web services integration. In *Proceedings of the International Conference on Database Systems for Advanced Applications*, number 3453 in LNCS, Beijing, China, April 2005. Springer Verlag. pages 27
- [37] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. A dense time semantics for web services specifications languages. In *Proceedings of the International Conference on Information and Communication Technologies : From Theory to Applications*, pages 647–652, Damascus, Syria, April 2004. IEEE. pages 31

-
- [38] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *Proceedings of the 6th International Conference on Enterprise Information Systems*, volume 4, pages 287–295, Porto, Portugal, April 2004. ICEIS Press. pages 31, 64, 65
- [39] D. Harel. Statecharts : a visual formalism for complex systems. *Science of computer programming, Elsevier Science Publishers*, 8 :231–274, 1987. pages 70, 120
- [40] E. N. Herness, R. J. High, and J. R. McGee. Websphere application server : A foundation for on demand computing. *IBM Systems Journal*, 43(2) :213–237, 2004. pages 36
- [41] C. Herring and Z. Milosevic. Implementing b2b contracts using biztalk. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2001. IEEE. pages 36
- [42] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to petri nets. In *Proceedings of the International Conference on Business Process Management*, number 3649 in LNCS, pages 220–235, Nancy,France, September 2005. Springer Verlag. pages 31, 32
- [43] M. A. Holliday, J. T. Houston, and E. M. Jones. From sockets and RMI to web services. In *Proceedings of the Technical Symposium on Computer Science Education*, pages 236–240, Portland, USA, March 2008. ACM. pages 8
- [44] <http://bpel.xml.org/>, last visite, June 2008. pages 6
- [45] <http://fr.bea.com/>. Bea systems, last visite, June 2008. pages 52
- [46] <http://rosettanet.org/>. Rosettanet overview cluster, segments, and pips, version 02.01.00. pages 43
- [47] <http://uddi.org/>. Universal description, discovery and integration. ibm, microsoft, hp, intel, sap,oasis., last visite, september 2008. pages 7
- [48] <http://www306.ibm.com/software/websphere/>. Websphere, last visite, June 2008. pages 36
- [49] <http://www.daml.org/services/>. Defense advanced research projects agency agent markup language-services, last visite, June 2008. pages 41, 42
- [50] http://www.daml.org/services/owl_s/. Ontology web language-services, last visite, June 2008. pages 41
- [51] <http://www.microsoft.com/france/biztalk/default.msp>x. Biztalk. pages 36
- [52] <http://www.opentravel.org/>. Ota open travel alliance, last visite, June 2008. pages 43

- [53] <http://www.w3.org/TR/soap/>, last visite, June 2008. pages 6
- [54] <http://www.w3.org/TR/wsci/>. Web service choreography interface, last visite, June 2008. pages 6
- [55] <http://www.w3.org/TR/wsdl>. Web service description language, last visite, June 2008. pages 6
- [56] <http://www.w3.org/TR/xpath>. Xml path language, last visite, June 2008. pages 27
- [57] <http://www.w3.org/TR/xquery/>. Xml query language, last visite, June 2008. pages 27
- [58] <http://www.w3.org/TR/xslt.html>. Xsl transformations, last visite, June 2008. pages 27
- [59] <http://www.xcbl.org/>. Xml common business library, last visite, June 2008. pages 129
- [60] <http://www.yawlsystem.com/>. Yet another workflow language, last visite, June 2008. pages 36
- [61] R. Hull, M. Benedikt, V. Christophides, and J. Su. EServices : A look behind the curtain. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–14, San Diego, California, USA, June 2003. ACM. pages 31, 35
- [62] G. Joeris and O. Herzog. Managing evolving workflow specifications. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems*, pages 310 – 321, New York, USA, August 1998. IEEE Computer Society. pages 57
- [63] B. Kalali, P. Alencar, and D. Cowan. A service-oriented monitoring registry. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, pages 107 – 121, Toronto, Ontario, Canada, October 2003. IBM Press. pages 54, 62, 63
- [64] P. Kaminski, H. Müller, and M. Litoiu. A design for adaptive web service evolution. In *Proceedings of the International workshop on Self-adaptation and self-managing systems (SEAMS)*, pages 86–92, Shanghai, China, May 2006. ACM Press. pages 53
- [65] P. Kaminski, H. Müller, and M. Litoiu. A design technique for evolving web services. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, pages 303–317, Toronto, Ontario, Canada, October 2006. ACM Press. pages 53, 62, 63
- [66] T. Kawamura, J.-A. De Blasio, T. Hasegawa, M. Paolucci, and K. Sycara. Preliminary report of public experiment of semantic service matchmaker with uddi business

registry. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, volume 2910 of *LNCS*, pages 208–224, Trento - Italy, December 2003. Springer. pages 42, 64, 65

- [67] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. Distributed, parallel web service orchestration using XSLT. In *Proceedings of the International Conference on e-Science and Grid Technologies*, pages 312–319, Melbourne, Australia, December 2005. IEEE Computer Society. pages 27
- [68] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. Compilation of xslt into dataflow graphs for web service composition. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, pages 141–149, Singapore, May 2006. IEEE Computer Society. pages 27
- [69] S. Kim. IT compliance of industrial information systems : Technology management and industrial engineering perspective. *Systems and Software, Elsevier Science*, 80(10) :1590–1593, 2007. pages 7
- [70] K. Koidl and O. Conlan. Engineering information systems towards facilitating scrutable and configurable adaptation. In *Proceedings of the International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, number 5149 in *LNCS*, pages 405–409, Hannover, Germany, July-August 2008. Springer. pages 7
- [71] M. Kradolfer and A. Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems*, pages 104–114, Edinburgh, Scotland, September 1999. IEEE Computer Society. pages 1, 17, 55, 62, 63, 64, 135
- [72] S. Krakowiak, M. Dumas, and M.-C. Fauvet. *Intergiciel et Construction d'Applications Réparties*. INRIA Rhône-Alpes et IMAG, 2006. pages 9, 11, 12, 13
- [73] J. M. Kuster, C. Gurth, A. Foster, and G. Engels. Detecting and resolving process model differences in absence of change log. In *Proceedings of the International Conference on Business Process Management*, number 5240 in *LNCS*, pages 244–260, Milano, Italy, September 2008. pages 37
- [74] N. Lohmann. Correcting deadlocking service choreographies using a simulation-based graph edit distance. In *Proceedings of the International Conference on Business Process Management*, number 5240 in *LNCS*, pages 132–147, Milano, Italy, September 2008. Springer Verlag. pages 17, 23, 39, 65, 66, 97, 98, 127, 131, 132, 136
- [75] Z. Maamar, Q. Z. Sheng, and D. Benslimane. Sustaining web services high-availability using communities. In *Proceedings of the International Conference on*

- Availability, Reliability and Security*, pages 834–841, Barcelona , Spain, March 2008. IEEE Computer Society. pages 40
- [76] A. Martens. Consistency between executable and abstract processes. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, EEE '05*, pages 60–67, Hong Kong, China, March 2005. IEEE Computer Society Press. pages 32
- [77] A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing compatibility of bpm processes. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW)*, pages 147–156, Guadeloupe, French Caribbean, February 2006. IEEE. pages 31, 32
- [78] M. Mecella, B. Pernici, and P. Craca. Compatibility of e-services in a cooperative multi-platform environment. In *Proceedings of the International Workshop on Technologies for E-Services*, volume 2193 of *LNCS*, pages 44–57, Rome, Italy, September 2001. pages 44, 62, 63
- [79] J. Mendling, B.F. van Dongen, and W.M.P. van der Aalst. Getting rid of the or-join in business process models. In *Proceedings of the Tenth IEEE International Enterprise Computing Conference (EDOC)*, pages 3–14, Annapolis Maryland, USA, October 2007. IEEE Computer Society. pages 31, 51
- [80] M. Moran, M. Zaremba, A. Mocan, and C. Bussler. Using WSMX to bind requester and provider at runtime when executing semantic web services. In *Proceedings of the WIW 2004 Workshop on WSMO Implementations*, volume 113, Frankfurt, Germany, September 2004. CEUR Workshop proceedings. pages 40, 64, 65
- [81] P. H. Morris, R. A. Gray, and R. E. Filman. Goto removal based on regular expressions. *Journal of Software Maintenance : Research and Practice, Wiley-Blackwell*, 9(1) :47 – 66, Jan.-Feb. 1997. pages 70, 147
- [82] 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*, pages 993–1002, Banff, Alberta, Canada, May 2007. ACM. pages 29, 65
- [83] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *Proceedings of the 29th International Conference on Software Engineering*, pages 54–64, Minneapolis USA, May 2007. IEEE Computer Society. pages 30, 35, 59, 70
- [84] P. Oaks and A.H.M. ter Hofstede. Guided interaction : A mechanism to enable ad hoc service interaction. *BPM Center Report BPM-05-12*, 2005. pages 58

-
- [85] N. Onose and J. Siméon. XQuery at your web service. In *Proceedings of the International Conference on World Wide Web*, pages 603–611, New York, NY, USA, May 2004. ACM. pages 27
- [86] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breuteland, M. Dumas, and A. H. M. ter Hofstede. WofBPEL : A tool for automated analysis of BPEL processes. In *Proceedings of the 3rd International Conference on Service-Oriented Computing - ICSOC*, number 3826 in LNCS, pages 484–489, Amsterdam, The Netherlands, December 2005. Springer Verlag. pages 32
- [87] S. Overhage and P. Thomas. WS-Specification : Specifying web services using UDDI improvements. In *Proceedings of the NetObjectDays NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, number 2593 in LNCS, pages 100–119, Erfurt, Germany, October 2002. Springer. pages 46
- [88] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the First International Semantic Web Conference (ISWC)*, volume 2342 of LNCS, pages 333 – 347, Sardinia, Italy, June 2002. Springer. pages 42, 64, 65
- [89] J. Pathak, S. Basu, and V. Honavar. Modeling web service composition using symbolic transition systems. In *Proceedings of the 21st Conference on Artificial Intelligence (AAAI-06) Workshop on AI-driven Technologies for Service-Oriented Computing*, pages 65–80, Boston, Massachusetts, July 2006. AAAI Press. pages 31, 33, 34, 44, 71
- [90] C. Peltz. Web services orchestration and choreography. *Computer, IEEE Computer Society*, 36(10) :46–52, 2003. pages 12, 30
- [91] J. Ponge. A new model for web services timed business protocols. In *Atelier SISW, INFORSID*, pages 15–28, Hammamet, Tunisie, Mai 2006. Hermès. pages 31
- [92] S. R. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Proceedings of the International Middleware Conference on Middleware, 5th ACM/IFIP/USENIX*, volume 3231 of LNCS, pages 331–351, Toronto, Canada, October 2004. Springer-Verlag. pages 58
- [93] M. Reichert and P. Dadam. ADEPT flex supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems, Springer*, 10(2) :93–129, 1998. pages 50
- [94] S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data and Knowledge Engineering, Elsevier*, 50 :9–34, 2004. pages 49

- [95] S. Rinderle, A. Wombacher, and M. Reichert. Evolution of process choreographies in DYCHOR. In *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS)*, LNCS 4275, pages 273–290, Montpellier, France, November 2006. pages 48, 62, 63, 136
- [96] A. SANFELIU and F. KING-SUN. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, 13(3) :353–362, 1983. pages 37
- [97] H. W. Schmidt and R. H. Reussner. Generating adapters for concurrent component protocol synchronisation. In *Proceedings of the IFIP TC6/WG6.1 , the Fifth IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 81, pages 213–229, Enschede, The Netherlands, March 2002. Springer-Verlag, Berlin. pages 61, 65, 66
- [98] A. Schönberger and G. Wirtz. Realising rosettanelt pip compositions as web service orchestrations. In *Proceedings of the International Conference on E-Learning, E-Business, Enterprise Information Systems, E-Government, & Outsourcing*, pages 141–147, Las Vegas, Nevada, USA, June 2006. CSREA Press. pages 12, 29
- [99] Z. Shen and J. Su. Web service discovery based on behavior signatures. In *Proceedings of the International Conference on Service Computing SCC'05 IEEE International Conference*, pages 279–286, Orlando, Florida, USA, July 2005. IEEE Computer Society Press. pages 30, 42, 59, 65
- [100] O. Sokolsky, S. Kannan, and I. Lee. Simulation-based graph similarity. In *Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920, pages 426–440, Vienna Austria, March 2006. Springer. pages 23, 31, 35, 38, 65, 66, 76, 97, 98, 127, 131, 132, 136
- [101] M. Stollberg, E. Cimpian, and D. Fensel. Mediating capabilities with delta-relations. In *Proceedings of the 1st International Workshop on Mediation in Semantic Web Services (Mediate 2005)*, Amsterdam, the Netherlands, December 2005. pages 41, 65
- [102] Y. Taher, D. Benslimane, M.C. Fauvet, and M.Z. Maamar. Towards an approach for web services substitution. In *Proceedings of the 10th IEEE International Database Engineering and Applications Symposium*, pages 220–235, Delhi, India, September 2006. IEEE. pages 30, 40, 65, 66
- [103] S. Tai, R. Khalaf, and T. Mikalsen. Composition of coordinated web services. In *Proceedings of the Middleware Conference, 5th ACM/IFP/USENIX International Conference on Middleware*, volume 78 of *Web services : composition, integration and*

interoperability, pages 294–310, Toronto, Canada, October 2004. Springer-Verlag, Berlin. pages 59

- [104] H. Tan and W. M. P. van der Aalst. Implementation of a YAWL work-list handler based on the resource patterns. In *Proceedings of the International Conference on Computer Supported Cooperative Work in Design*, pages 1184–1189, Nanjing, China, May 2006. IEEE. pages 36
- [105] F. L. Tiplea and A. Tiplea. Instantiating nets and their applications to workflow nets. In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pages 367–373, Timisoara, Romania, September 2005. pages 45
- [106] W.-J. van den Heuvel, H. Weigand, and M. Hiel. Configurable adapters : the substrate of self-adaptive web services. In *Proceedings of the International Conference on Electronic Commerce : The Wireless World of Electronic Commerce*, number 258 in ACM International Conference Proceeding Series, pages 127–134, Minneapolis, MN, USA, August 2007. ACM. pages 29
- [107] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL : yet another workflow language. *Information Systems, Elsevier*, 30(4) :245–275, 2005. pages 36
- [108] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases, Springer*, 14(1) :5–51, 2003. pages 30
- [109] W.M.P. van der Aalst. Generic workflow models : how to handle dynamic change and capture management information ? In *Proceedings of the Fourth IFCS International Conference on Cooperative Information Systems*, pages 115–126, Edinburgh, Scotland, September 1999. pages 56, 61, 63, 64, 135
- [110] W.M.P. van der Aalst and T. Basten. Inheritance of workflows : An approach to tackling problems related to change. *Journal of Theoretical Computer Science, Elsevier*, 24 :125–203, 2002. pages 51, 62, 63, 64
- [111] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3) :555–600, May 1996. pages 70
- [112] M. Vukmirovic, M. Gawinecki, P. Kobzdej, M. Ganzha, and M. Paprzycki. Implementing message exchange between Airlines' GDSs and travel systems with ontologically demarcated data. In *Proceedings of the International Conference on Information Technology Interfaces*, pages 463–468, Cavtat, Dubrovnik, Croatia, June 2007. IEEE. pages 29
- [113] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1) :168–173, January 1974. pages 37, 38

- [114] B. Weber, S. Rinderle, and M. Reichert. Change patterns and change support features in process-aware information systems. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE)*, number 4495 in LNCS, pages 574–588, Trondheim, Norway, June 2007. Springer. pages 17, 46, 63, 64, 136
- [115] R. Weinreich, T. Ziebermayr, and D. Draheim. A versioning model for enterprise services. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINA '07)*, volume 2, pages 570–575, Niagara Falls, Canada, May 2007. IEEE Computer Society. pages 52, 62, 63, 64
- [116] P. Wohed, W. M.P van der Aalst, M. Dumas, and A. H.M. ter Hofstede. Analysis of web services composition languages : The case of BPEL4WS. In *Proceedings of the International Conference on Conceptual Modeling*, pages 200–215, Chicago, USA, November 2003. Springer-Verlag, Berlin. pages 11, 30
- [117] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. In *Proceedings of the IEEE International Conference on Multimedia and Expo, ICME 2004*, pages 359–368, Taipei, Taiwan, March 2004. IEEE Computer Society Press. pages 31, 44, 64, 65, 71, 130, 136
- [118] A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming BPEL into annotated deterministic finite state automata for service discovery. In *Proceedings of the IEEE International Conference on Web Services*, pages 316–323, San Diego, USA, July 2004. pages 30
- [119] A. Wombacher, B. Mahleko, and E. Neuhold. IPSI-PF : a business process matchmaking engine. In *Proceedings of the International Conference on Electronic Commerce (CEC)*, pages 137–145, San Diego, California, USA, July 2004. pages 45, 63
- [120] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(2) :292–333, March 1997. pages 22, 27, 29, 60
- [121] H. Yu, J. E. Moreira, P. Dube, I. Chung, and L. Zhang. Performance studies of a websphere application, trade, in scale-out and scale-up environments. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–8, Long Beach, California, USA, March 2007. IEEE. pages 36
- [122] J. Zdravkovic and P. Johanesson. Cooperation of processes through message level agreement. In *Proceedings of the 16th CAiSE International Conference*, number 3084 in LNCS, pages 564–579, Riga, Latvia, June 2004. Springer Verlag. pages 27

Résumé

La technologie des services web est aujourd'hui largement utilisée comme support de l'interopérabilité entre les applications. Dans ce cadre, les interactions entre deux applications encapsulées par des services web sont réalisées par le biais d'un ensemble d'échanges de messages appelé *conversation*. Une conversation peut échouer parce que l'interface fournie d'un participant a été modifiée et n'est plus compatible avec celle requise par l'autre participant. L'étude rapportée dans cette thèse porte sur la réconciliation de telles conversations. La solution proposée est le canevas *ArchiMed*.

Dans un premier temps, une modélisation des interfaces comportementales par des automates est adoptée. Dans cette modélisation, seul le comportement externe (comportement observable) est considéré. En d'autres termes, il n'y a que les opérations d'envoi et de réception de messages qui sont décrites dans les interfaces. Une fois les interfaces comportementales décrites en automates, une étape de détection des incompatibilités entre les différentes définitions de l'interface d'un service est réalisée. La détection des incompatibilités est automatique et portent sur des changements élémentaires dans les interfaces qui sont : *l'ajout, la suppression et la modification* d'opérations. Une visualisation des incompatibilités entre les interfaces est rendue disponible.

Le canevas maintient les descriptions des versions successives de l'interface du service ainsi que l'ensemble minimal de médiateurs pour que les clients puissent interagir avec le service au travers de l'une de ses versions. A l'initiation d'une conversation par un client, si nécessaire, le canevas sélectionne parmi les médiateurs disponibles celui qui résout les incompatibilités entre le client et le service. La sélection du médiateur est basée sur une mesure de similarité entre les interfaces afin de découvrir la description de l'interface fournie qui est conforme avec l'interface requise du client.

La validation expérimentale du canevas, a été effectuée par le traitement d'une collection de tests qui contient les descriptions des interfaces comportementales des services. Une étude quantitative et comparative à des travaux similaires est réalisée et montre l'apport significatif de notre proposition.

Mots-clés: Services web, interface, fournie, requise, structure, comportement, conversation, automate, médiation, incompatibilité, simulation, détection, résolution.

Abstract

As Web service interactions rely on message exchanges, modelling Web service aims at describing messages as well from the structural point of view (types of exchanged messages) as from the behavioural point of view (control flow between message exchanges). With this setting, a Web service's interface is defined as the set of messages it can receive and send, and the inter-dependencies between these messages. We distinguish the provided interface an existing service exposes, from its required interface as it is expected by its clients (i.e. applications). When a Web service evolves, its interface is more likely to be modified too. This leads to the situation where the provided interface of a service does not correspond any more to the one its partners expect. In this thesis, we introduce our techniques for the detection and the resolution of incompatibilities. Our contribution is the framework, namely *ArchiMed*.

We analyse successive versions of a service interface in order to detect changes that cause clients using any of the earlier versions not to interact properly with a later version. We focus on behavioural incompatibilities and adopt the notion of simulation as a basis for determining if a new version of a service is behaviourally compatible with a previous one. Unlike prior work, our technique does not simply check if the new version of the service simulates the previous one. Instead, having identified one source of incompatibility, the technique goes on to identify other incompatibilities, thus providing more detailed diagnostics. The technique has been implemented in a framework that visually pinpoints a set of changes that cause one behavioural interface not to simulate another one.

The framework, has been tested on a collection of interfaces corresponding to standard business-to-business choreographies. To resolve these incompatibilities, two solutions thus apply : (1) modify the service in order to make the interface it provides match the interface required by each client ; (2) introduce an adapter that reconciles the provided interface with those required by the partners. The former solution is not satisfying because the same service may interact with many other partners which consider its original interface. The same service has to expose as many provided interfaces as collaborations it is involved in. The latter solution consists in supplying a mediator which is capable of matchmaking each of the required interfaces with the provided interface. The selection of the suitable mediator is based on the similarity measure between the provided interface an the required interface by the client.

Keywords: Web services, interface, provided, required, structure, behaviour, conversation, automaton, mediation, incompatibility, simulation, detection, resolution.