Composition d'applications et de leurs Interfaces homme-machine dirigée par la composition fonctionnelle

Cédric Joffroy

UNIVERSITÉ DE NICE–SOPHIA ANTIPOLIS — UFR Sciences
École Doctorale de Sciences et Technologies de l'Information
et de la Communication (STIC)

T H È S E

pour obtenir le titre de
Docteur en Sciences
de l'UNIVERSITÉ de Nice–Sophia Antipolis

Discipline : Informatique

présentée et soutenue par
Cédric JOFFROY

COMPOSITION D'APPLICATIONS ET DE LEURS INTERFACES HOMME-MACHINE DIRIGÉE PAR LA COMPOSITION FONCTIONNELLE

Thèse dirigée par Anne-Marie DERY–PINNA et Michel RIVEILL
soutenue le 06 – 06 – 2011

Jury :

M.	BASTIDE Rémi	Professeur des Universités	Rapporteur
Mme.	BLAY–FORNARINO Mireille	Professeur des Universités	Examinateur
Mme.	CALVARY Gaëlle	Professeur des Universités	Rapporteur
Mme.	DERY–PINNA Anne–Marie	Maître de Conférence	Co-Directeur
Mme.	DUPUY–CHESSA Sophie	Maître de Conférence	Examinateur
M.	RIVEILL Michel	Professeur des Universités	Directeur

A ma petite sœur Cindy

Remerciements

Je tiens tout d'abord à remercier Anne–Marie Dery–Pinna et Michel Riveill d'avoir accepté d'être mes directeurs et de m'avoir donné l'opportunité d'effectuer cette thèse. Sans vous, je ne serais peut-être pas allé au bout de cette thèse. Merci également à vous deux, à Mireille Blay–Fornarino et Audrey Occello d'avoir relu la thèse. Cela a permis de grandement améliorer le document qui ne serait pas ce qu'il est sans vos retours et commentaires constructifs.

Je tiens à remercier à nouveau Anne–Marie d'avoir su me supporter durant plus de trois ans avec les hauts et les bas qu'il y a pu avoir, de m'avoir laissé la liberté dans mon travail mais également de m'avoir proposé ce sujet de thèse qui était et reste un sujet passionnant où beaucoup de pistes restent encore à creuser. Sans nul doute, c'est une problématique d'avenir qui saura très certainement trouver sa place dans nos applications de demain.

Je remercie Gaëlle Calvary et Rémi Bastide d'avoir accepté la lourde tâche de rapporteur. Merci pour vos retours et commentaires. La voie ouverte par vos questions montre que le travail peut être poursuivi et qu'il sera intéressant d'approfondir certains points ou d'attaquer le problème d'un autre point de vue. Je remercie encore Gaëlle pour nos discussions durant les réunions de travail, conférences et autres cours et pour sa joie de vivre. En espérant, un jour, que l'on puisse travailler ensemble.

Je remercie également Sophie Dupuy–Chessa et Mireille Blay–Fornarino d'avoir accepté de participer à mon jury.

Merci à l'ensemble des membres des équipes RAINBOW et MODALIS pour l'accueil qui m'a été fait, pour la joie de vivre qui y règnent, les discussions autour d'un café qui ont fait émerger des idées, sans oublier les journées du pôle faites dans des lieux magnifiques et les activités qui ont pu être organisées…

Merci à l'ensemble des thésards avec qui j'ai eu la possibilité de passer ces années riches en échanges et partages. Merci à Javier pour nos nombreuses discussions sur la vie et à Ketan de nous avoir supportés. Merci à Clémentine pour nos petites (grandes) pauses thé, pour nos weekend à l'école à travailler et pour les bons moments passés. Merci à Tram pour sa gentillesse et sa disponibilité.

Merci à l'ensemble de l'équipe enseignante et administrative de Polytech Nice Sophia. Ces années de cours et d'enseignements n'auraient pas été les mêmes sans vous. Merci à Marc Gaëtano de m'avoir donné l'opportunité d'enseigner dans sa matière, à Audrey avec qui j'ai eu l'occasion de participer à la création d'un cours, à Marie–Hélène pour son sourire et sa bonne humeur permanents, à Jérémy pour tous les services qu'il a pu me rendre…

Enfin, et pour conclure, je tiens à remercier mes parents et ma sœur qui m'ont soutenu et ont su trouver les mots pour que je puisse arriver au bout. Sans eux, je ne serais pas là où j'en suis maintenant. Et je remercie également Michele qui a su m'encourager dans la dernière ligne droite et être présente à mes côtés.

Table des matières

1 Introduction **15**

2 Scénarios de compositions fonctionnelles **19**
2.1 Applications de l'étude de cas 20
2.2 Composition au niveau des IHM et problèmes sous-jacents 24
2.3 Compositions fonctionnelles : réponse aux ambiguïtés de la composition d'IHM . 28
2.4 Conclusion 37

3 Domaine d'étude : la composition d'applications **39**
3.1 La composition au niveau fonctionnel 40
3.2 La composition au niveau des IHM 56
3.3 La composition d'applications 68
3.4 Synthèse de l'état de l'art 78

4 Processus de composition d'applications **81**
4.1 Démarche générale du processus de composition 82
4.2 Formalisation des entrées et du résultat de la composition 84
4.3 Conclusion 117

5 Validation de l'approche `Alias` **119**
5.1 Les transformations au sein du processus de composition 120
5.2 Atelier d'aide à la composition d'IHM 134
5.3 Conclusion 142

6 Conclusion et perspectives **143**
6.1 Conclusion 143
6.2 Perspectives 145

A Description des applications et des compositions **157**

B Moteur de composition **171**

C Règles de transformations et méta-modèles `eCore` **197**

Table des figures

2.1 IHM associées au *Web Service BusinessService* . . . 22
2.2 IHM associées au *Web Service SocialInsuranceService* . . . 24
2.3 Mise en évidence des informations communes entre les deux IHM . . . 25
2.4 Juxtaposition des fonctionnalités impliquant une redondance d'information . . . 26
2.5 Résolution de la redondance d'information par union des courriels *Email* . . . 26
2.6 Résolution de redondance d'information par sélection . . . 26
2.7 Résolution de redondance d'information par union sans redondance . . . 27
2.8 Utilisation de deux sorties comme entrée dans une autre IHM . . . 27
2.9 Suppression d'un des champs de saisi remplacé par l'utilisation de valeurs . . . 28
2.10 *Workflow* de la première composition avec exécution en parallèle . . . 29
2.11 Affectation des paramètres d'entrée dans une activité *assign* . . . 30
2.12 Affectation des paramètres de sortie dans une activité *assign* . . . 30
2.13 *Workflow* de la seconde composition avec exécution en séquence . . . 32
2.14 Affectation du paramètre d'entrée pour l'opération *getByCard* . . . 32
2.15 Calcule du paramètre d'entrée de l'opération *getBusinessInfo* . . . 33
2.16 Affectation des paramètres de sortie où certains sont sélectionnés . . . 33
2.17 *Workflow* de la troisième composition avec exécution en parallèle . . . 35
2.18 Affectation des paramètres d'entrée . . . 35
2.19 Affectation du paramètre de retour avec fusion . . . 36

3.1 Représentation de l'assemblage de composants au sein d'UML2 . . . 48
3.2 Composant FRACTAL du noyau fonctionnel de l'application *Business* . . . 49
3.3 Composant SCA du noyau fonctionnel de l'application *Business* . . . 51
3.4 Assemblage de composants FRACTAL . . . 53
3.5 Assemblage de composants SCA . . . 54
3.6 Rendu Swing de la description SUNML . . . 58
3.7 Opérateurs de type unaire présents dans `Amusing` . . . 62
3.8 Opérateurs de type binaire présents dans `Amusing` . . . 62
3.9 Ensemble des opérations de compositions présentes dans `ComposiXML` . . . 63
3.10 Exemple d'arbre de tâches décrit avec `CTTe` . . . 65
3.11 Modèle de tâches simplifié qui décrit la saisie du nom et du nuémro de carte . . . 66
3.12 Annotation du modèle de tâches avec les fragments d'IHM . . . 67
3.13 Deux applications juxtaposées au sein d'*iGoogle* . . . 69
3.14 Création d'un *pipe* qui utilise deux *Web Services* pour obtenir les informations d'une personne . . . 70
3.15 Modèle de tâches dérivé des tâches annotées correspondant à l'utilisation de l'une ou l'autre des opérations . . . 73
3.16 Diagramme de séquence représentant le *workflow* de l'IHM pour obtenir les informations d'un employé . . . 75
3.17 IHM pour l'obtention des informations d'un employé . . . 75
3.18 Modèle d'application correspond à l'obtention des informations des deux services 76

4.1 Démarche globale pour la réalisation de la composition d'applications 84
4.2 Méta-modèle `AliasComponent`dans son intégralité 85
4.3 Méta-modèle `AliasComponent`de l'application 87
4.4 Représentation du composant fonctionnel au sein du méta-modèle `AliasComponent` 89
4.5 Instantiation du service *BusinessService* au sein du méta-modèle `AliasComponent` . 91
4.6 Représentation du composant d'IHM au sein du méta-modèle `AliasComponent` . . 93
4.7 IHM pour l'opération `getBusinessInfo` 94
4.8 Instantiation de l'IHM `getBusinessInfoUI` au sein du méta-modèle `AliasComponent` 95
4.9 IHM pour l'opération `getAddresses` 96
4.10 Illustration d'un extrait de l'application *Business* avec `AliasComponent` 98
4.11 Représentation de la composition fonctionnelle au sein d'`AliasComponent` 103
4.12 Exécution en parallèle des deux opérations *getAddresses* et *getByCard* 105
4.13 Affectation des paramètres d'entrée 105
4.14 Fusion des paramètres de sorties des deux opérations 106
4.15 Instantiation de la composition fonctionnelle au sein d'`AliasComponent` 108
4.16 Méta-modèle `AliasComponent`dans son intégralité 110

5.1 Extrait du WSDL du *Web Service BusinessService* 123
5.2 Abstraction du service *BusinessService* au sein du méta-modèle `AliasComponent` . . 124
5.3 Méta-modèle de MXML 125
5.4 Méta-modèle de MXML 126
5.5 Résultat de l'abstraction en utilisant la table d'abstraction 127
5.6 Résultat de l'abstraction de l'application `Business` au sein d'`AliasComponent` . . . 129
5.7 Méta-modèle de BPEL 130
5.8 IHM principale de l'atelier 134
5.9 Remontée des conflits 135
5.10 Résolution d'un conflit associé à une action 135
5.11 Résolution d'un conflit associé à une sortie 136
5.12 Assemblage de composants correspondant au résultat de la composition 137
5.13 Rendu de l'IHM en `Java` Swing 137
5.14 Radar des résultats des tests utilisateurs 141

C.1 Méta-modèle `eCore` de WSDL 198
C.2 Méta-modèle `eCore` d'`AliasComponent` 199

Listings

2.1 Extrait du schéma de données du businessInformation . . . 21
2.2 Déclaration de l'opération getBusinessInfo . . . 21
2.3 Extrait du schéma de données associé à l'opération getBusinessInfo . . . 21
2.4 Déclaration de l'opération getAddresses . . . 21
2.5 Extrait du schéma de données associé à l'opération getAddresses . . . 22
2.6 Extrait du schéma de données du socialInsuranceInformation . . . 23
3.1 Extrait du WSDL du service BusinessService pour l'opération *getBusinessInfo* . . . 41
3.2 Schéma de données pour le Web Service BusinessService restreint à l'opération *getBusinessInfo* . . . 42
3.3 Exemple de message SOAP pour la demande et la réponse de l'opération *getBusinessInfo* . . . 42
3.4 Extrait de la description OWL-S pour l'opération *getBusinessInfo* . . . 44
3.5 Extrait du WSDL avec les annotations OWL-S pour l'opération *getBusinessInfo* . . . 44
3.6 Version simplifié du BPEL avec la description du flot de données et de contrôle . . . 45
3.7 Processus composite correspondant à l'exécution en parallèle des deux opérations 47
3.8 Interface fournies par le composant Business . . . 49
3.9 Implémentation de l'interface du composant Business . . . 49
3.10 Implémentation du composant Business . . . 50
3.11 Implémentation du composant Business dans WCOMP . . . 51
3.12 Interface fournies par le composant Emergency . . . 53
3.13 Extrait de l'implémentation de l'interface du composant Emergency . . . 53
3.14 Description de l'assemblage de composants correspondant à la figure 3.4 . . . 53
3.15 Description de l'assemblage de composants correspondant à la figure 3.5 . . . 54
3.16 Description de l'assemblage de composants au sein de WCOMP . . . 55
3.17 Description SUNML de l'IHM pour l'opération getBusinessInfo . . . 58
3.18 Description de l'IHM Abstraite USIXML de l'opération getBusinessInfo . . . 59
3.19 Description de l'IHM Concrète USIXML de l'opération getBusinessInfo . . . 60
3.20 Fichier comportant les annotations du *Web Service* de l'application *Business* par des éléments d'IHM . . . 71
3.21 Profile du *workflow* de l'IHM pour l'obtention des infos d'un employé . . . 74
4.1 Règle OCL qui contraint l'attribut *cardinality* sur les ports de données en entrée et sortie . . . 89
4.2 Règle OCL qui contraint l'attribut *cardinality* sur les ports de données en entrée et sortie . . . 93
4.3 Règle OCL qui contraint l'attribut *selection* sur le port d'entrée . . . 93
4.4 Règle OCL qui décrit les liens entre deux entrées . . . 96
4.5 Règle OCL qui décrit les liens entre deux sorties . . . 96
4.6 Règle OCL qui décrit les liens entre un *Event* et une *Action* . . . 96
4.7 Règle OCL qui décrit les liens entre deux entrées . . . 103
4.8 Règle OCL qui décrit les liens entre deux sorties . . . 104
4.9 Règle OCL qui décrit les liens entre une sortie et une entrée . . . 104
4.10 Règle OCL qui décrit les liens entre deux actions . . . 104

4.11 Règle OCL qui décrit les liens internes à un composant d'IHM 109
5.1 *helpers* mis en place pour aider à la transformation 121
5.2 Règle principale qui permet le déclenchement de la transformation 121
5.3 Règle principale qui permet le déclenchement de la transformation 121
5.4 Règle principale qui permet le déclenchement de la transformation 122
5.5 Description de l'IHM pour l'opération *getBusinessInfo* 125
5.6 Liens d'interaction entre la partie IHM et la partie fonctionnelle 128
5.7 Concaténation de deux variables pour fournir une entrée à la seconde opération . 131
A.1 WSDL pour le service de `SocialInsuranceService` 157
A.2 Schéma de données pour le Web Service `SocialInsuranceService` 158
A.3 IHM associée à l'opération `getByCard` 159
A.4 WSDL pour le service de `BusinessService` 160
A.5 Schéma de données pour le Web Service `BusinessService` 160
A.6 IHM associée à l'opération `getBusinessInfo` 161
A.7 XSD de la $1^{ème}$ composition 162
A.8 WSDL de la $1^{ème}$ composition 163
A.9 BPEL de la $1^{ème}$ composition 163
A.10 XSD de la 2^{nde} composition 165
A.11 WSDL de la 2^{nde} composition 165
A.12 BPEL de la 2^{nde} composition 166
A.13 XSD de la $3^{ème}$ composition 167
A.14 WSDL de la $3^{ème}$ composition 168
A.15 BPEL de la $3^{ème}$ composition 169
B.1 Faits `Prolog` représentant le composant fonctionnel 172
B.2 Faits `Prolog` représentant la définition d'un port de données 172
B.3 Faits `Prolog` représentant la définition d'un port d'action 172
B.4 Faits `Prolog` représentant le composant d'IHM 173
B.5 Faits `Prolog` représentant le port d'entrée du composant d'IHM 173
B.6 Faits `Prolog` représentant un port de sortie du composant d'IHM 173
B.7 Faits `Prolog` représentant le port d'événement du composant d'IHM 173
B.8 Faits `Prolog` représentant l'association entre composant fonctionnel et composant d'IHM 174
B.9 Faits `Prolog` représentant un lien de données entre deux entrées 174
B.10 Faits `Prolog` représentant un lien de données entre des sorties 174
B.11 Faits `Prolog` représentant un lien d'événement 174
B.12 Faits `Prolog` représentant le composant composite 175
B.13 Faits `Prolog` représentant la définition d'un port de données 175
B.14 Faits `Prolog` représentant la définition du port d'action 176
B.15 Faits `Prolog` représentant la définition des liens entre entrées du composant composite et un des sous-composants 176
B.16 Faits `Prolog` représentant la définition des liens entre sorties du composant composite et un des sous-composants 177
B.17 Faits `Prolog` représentant la définition des liens entre actions du composant composite et un des sous-composants 177
B.18 Prédicats permettants d'obtenir les premiers ou les seconds éléments d'une liste de couple 178
B.19 Prédicat pour obtenir la liste des entrées présentées 178
B.20 Prédicat pour obtenir la liste des sorties présentées 179
B.21 Prédicat pour obtenir la liste des actions présentées 179
B.22 Prédicats permettants d'obtenir la liste qui associe à chaque port la liste des ports des sous-composants auquel il est reliés (dans le cas des entrées) 180

B.23 Liste qui associe à un sous-composant l'ensemble de ses ports qui sont liés à un même port du composant composite . . . 180
B.24 Liste qui contient l'ensemble des ports liés à un port du composant composite pour un sous-composant donné . . . 181
B.25 Prédicat qui permet d'associer à chaque port présenté d'un sous-composant, sa représentation graphique . . . 181
B.26 Prédicat qui permet d'associer à un port sa représentation graphique . . . 182
B.27 Prédicat qui permet d'obtenir la représentation graphique . . . 182
B.28 Prédicat qui extrait les informations de la représentation graphique . . . 183
B.29 Prédicat permettant d'obtenir une liste qui associe à chacun des ports d'entrée du composant composite l'ensemble des représentation graphique . . . 183
B.30 Prédicat permettant d'obtenir une liste de l'ensemble des représentation graphique 184
B.31 Prédicat permettant d'obtenir une représentation graphique . . . 184
B.32 Prédicat permettant d'obtenir la représentation graphique . . . 184
B.33 Prédicats qui permettent d'obtenir la liste des points de fusion . . . 185
B.34 Prédicat qui permet de séparer une liste associant port à représentation graphique en deux sous-listes : une avec les points de fusion et une sans . . . 186
B.35 Prédicat qui permet de trouver le bon élément dans une liste de couple . . . 187
B.36 Prédicats qui permettent de tester si deux éléments sont identiques . . . 187
B.37 Prédicat qui réalise une comparaison de chacun des éléments deux à deux . . . 188
B.38 Prédicat qui permet de réaliser la comparaison de chacun des éléments . . . 188
B.39 Prédicat qui permet de détecter les points de conflits et qui fait appel au prédicat de comparaison . . . 189
B.40 Prédicat qui permet de rechercher les éléments qui sont cohérent avec le port fonctionnel . . . 191
B.41 Prédicat qui lance la création de l'ensemble des composants d'IHM . . . 192
B.42 Prédicat qui réalise la création des composants d'IHM . . . 193
B.43 Création des ports d'entrée . . . 194
B.44 Création de la définition des ports d'entrée . . . 194
B.45 Création des liens internes . . . 194
B.46 Création des liens de données . . . 195
B.47 Création des ports d'événement et des liens associés . . . 195
B.48 Création de l'ensemble des ports d'événement . . . 195
B.49 Création des liens d'événement . . . 196
C.1 Règles de transformation de modèles pour l'abstraction . . . 200

1 Introduction

Contexte et enjeux

La réutilisation d'applications n'est pas un problème récent. En effet, en 1983 se tenait le premier atelier en *Software Reuse*. Depuis, des normes et des architectures, telles que les services ou bien encore les composants sur étagère, ont permis de faciliter cette réutilisation au niveau fonctionnel. Cette réutilisation s'effectue au travers de la composition de ces services et composants permettant ainsi de créer les fonctionnalités désirées. En parallèle de ces approches dédiées au fonctionnel sont apparues des considérations similaires au niveau des Interfaces Homme–Machine (IHM). Ainsi, des langages dédiés à la description des IHM (*e.g.* USIXML, SUNML, ou bien encore XAML) réalisent la composition de celles-ci. Ce n'est que récemment que des approches mixtes sont apparues pour permettre la composition d'applications (*i.e.* qu'elles considèrent à la fois la partie fonctionnelle, l'IHM et les interactions entre les deux). C'est le cas par exemple avec les *Mashups* qui donnent la possibilité à des utilisateurs non experts de créer par juxtaposition d'applications un *bureau* regroupant l'ensemble des fonctionnalités dont ils ont besoin.

Dans le cas des développeurs, la démocratisation de l'utilisation des *Web Services* ou des composants a favorisé la production d'outils pour réaliser la composition de ceux-ci. Que ce soit au travers de l'utilisation d'un éditeur BPEL directement intégré au sein de NetBeans ou bien encore de plugins Eclipse pour faire de l'assemblage de composants, le développeur a un éventail large d'outils pour la création de noyaux fonctionnels (NF). Pour autant, ces outils permettent la réutilisation et la composition de la partie fonctionnelle d'une application, mais l'IHM n'est pas prise en compte. Le développeur doit alors, en plus de réaliser la composition fonctionnelle, créer une nouvelle IHM sans pouvoir réutiliser les IHM existantes déjà associées aux composants/services utilisés. Il perd ainsi le travail précédemment réalisé. Plusieurs possibilités lui sont offertes pour réaliser la nouvelle IHM que ce soit à l'aide d'outils de génération d'IHM à partir de la description de services ou bien tout simplement en repartant de zéro. L'inconvénient ici est de perdre la connaissance que l'utilisateur final avait des applications et de ne pas exploiter pour les IHM ce que l'on sait de la composition fonctionnelle.

Des travaux, comme par exemple les *Mashups*, `SOAUI` ou l'intégration de fonctionnalités volatiles, considèrent l'ensemble d'une application lors de la composition. Certaines approches réalisent la composition d'applications dirigée par la composition fonctionnelle, c'est le cas avec

SOAUI [THEC08] et les fonctionnalités volatiles [GRUD07]. Mais ces approches ne tirent pas pleinement partie des possibilités offertes par la composition fonctionnelle telles que la composition d'éléments hétérogènes, la possibilité d'exprimer des échanges, partages ou fusion de données ou bien encore de décrire différents types d'exécution (en parallèle ou en séquence, ou bien encore des conditions). En effet, ces approches nécessitent un développement spécifique contraignant et dédié à une technologie donnée sans pouvoir s'adapter à une autre et ne permettent que la juxtaposition d'IHM. Les autres approches qui réalisent la composition d'applications dirigées par la composition d'IHM ne permettent que de juxtaposer des applications (c'est le cas par exemple des *Mashups*) ou de décrire des échanges de données entre *widgets* pour permettre un remplissage automatique.

L'objectif de cette thèse est de faciliter le travail du développeur en lui permettant de réutiliser l'intégralité des applications à composer, c'est-à-dire, la partie fonctionnelle, l'IHM et les interactions entre l'IHM et la partie fonctionnelle. On souhaite également que le développeur puisse continuer à utiliser les outils dont il dispose pour composer les parties fonctionnelles des applications. Cette composition fonctionnelle guide la composition d'IHM qui se fera ainsi sur les éléments mis en évidence par celle-ci.

Contributions de la thèse

Cette thèse se trouve à la confluence de plusieurs domaines de recherche que sont : (i) le domaine des IHM, (ii) le domaine du logiciel et (iii) l'Ingénierie des Modèles. Dans le domaine des IHM, on s'intéresse plus spécifiquement aux travaux qui visent à la réutilisation des IHM par l'usage de descriptions abstraites des IHM et de la composition. Dans le domaine du logiciel, on s'intéresse aux travaux qui facilitent la réutilisation et la composition au travers de l'utilisation de services ou de composants par exemple. Enfin dans le domaine de l'Ingénierie des Modèles, on s'intéresse à travailler à un niveau modèle dans le but de faciliter la réutilisation de travaux effectués sur différentes plates-formes.

Les contributions de cette thèse sont à placer selon ces trois domaines avec en filigrane la composition d'applications comme objectif final. Elles comprennent essentiellement les points suivants :

- Une étude des travaux menés sur la composition qui prennent en considération la partie fonctionnelle, l'IHM ou l'intégralité d'une application ;
- Un processus de composition qui facilite la réutilisation des applications existantes selon les axes IHM et fonctionnalités.
- Un méta-modèle, `AliasComponent`, qui permet une composition indépendante d'une technologie donnée et qui modélise l'ensemble des éléments des applications à composer (IHM, fonctionnalités et interactions entre les deux).
- Un moteur de composition guidé par le développeur qui déduit la composition d'IHM en partant des modèles des applications et de la composition fonctionnelle. Le moteur de composition détecte les choix non "automatisables" (appelés conflits) et demande au développeur de choisir une solution pertinente.

Pour cela, chaque application est décrite à un niveau modèle et on conserve de celle-ci : (i) les fonctions proposées par le NF, (ii) les éléments d'IHM qui interagissent avec la partie fonctionnelle et (iii) l'ensemble des liens d'interactions (échanges de données ou émetteur d'événements) entre l'IHM et le NF.

La composition fonctionnelle est également décrite à un niveau modèle. On conserve de celle-ci : (i) les fonctions qu'elle propose, (ii) l'utilisation qui est faite des données (flot de données) afin de connaître les fusions, partages et échanges et (iii) l'utilisation qui est faite des opérations (flot de contrôle) afin de connaître comment celles-ci s'enchaînent.

Le méta-modèle `AliasComponent` permet de raisonner indépendamment d'une technologie

donnée. Il est implémenté par un moteur de composition qui déduit l'ensemble des éléments d'IHM à conserver ainsi que l'assemblage entre les IHM résultantes et la composition fonctionnelle qui doit être fait pour permettre de réaliser l'application finale. Enfin, le processus global de composition d'applications proposé dans cette approche est outillé dans un atelier nommé `Alias`. Cet atelier permet au développeur de résoudre les éventuels conflits et de visualiser le résultat obtenu afin de le contrôler. Il a été développé en `Prolog` (pour le moteur de composition) et est interfacé avec `Java`. Des tests utilisateurs ont été réalisés afin de valider l'approche et plus particulièrement la pertinence de la détection et résolution de conflits.

Plan de la thèse

Ce document est composé de six chapitres. Le chapitre 2 présente une étude de cas qui s'appuie sur deux applications. La composition réalisée sur ces deux applications a pour objectif de fournir une nouvelle application d'aide aux urgences. Il illustre la composition d'un point de vue IHM puis fonctionnelle en mettant en évidence qu'il peut être intéressant de s'appuyer sur la composition fonctionnelle pour réaliser la composition d'IHM.

Le chapitre 3 présente différents travaux sur le domaine de la composition, que ce soit la composition fonctionnelle, la composition d'IHM et plus particulièrement la composition d'applications. Cette étude permet de constater l'apport de la composition fonctionnelle quant à la prise de décision pour la composition d'IHM. Cette étude démontre également le besoin de s'appuyer sur une architecture particulière pour permettre la réalisation ainsi que le besoin de travailler à un niveau d'abstraction qui permette de réaliser la composition d'applications indépendamment d'une technologie donnée. Les travaux sur la composition d'applications montrent quant à eux les solutions actuelles. Cet état de l'art permet de motiver et justifier la pertinence de nos travaux ainsi que leur nécessité.

Les chapitres 4 et 5 correspondent aux travaux réalisés durant cette thèse. Le chapitre 4 présente `Alias` qui correspond à la démarche mise en œuvre pour réaliser la composition d'applications (plus particulièrement les IHM et les interactions) dirigée par la composition fonctionnelle. Cette démarche s'appuie sur l'utilisation d'une formalisation du processus de composition qui permet de déduire les éléments d'IHM à conserver, de détecter des conflits et de construire l'application finale avec l'ensemble des interactions. Elle s'appuie également sur la méta-modélisation des applications à composer qui repose sur une représentation à base de composants permettant ainsi de distinguer l'ensemble des parties qui composent une application.

Le chapitre 5 présente la mise en œuvre qui a été faite de la démarche proposée dans le chapitre 4. Dans un premier temps, il décrit l'ensemble des transformations réalisé pour permettre l'abstraction des applications existantes ainsi que de la composition fonctionnelle mais également les transformations de concrétisation afin d'obtenir un rendu des obtenues par le processus de composition. Dans un second temps, il présente l'atelier d'aide à la composition, sur lequel des tests utilisateurs ont été effectués, qui a été développé afin de valider l'approche.

Pour finir, le chapitre 6 établit les conclusions de ces travaux de thèse ainsi que les perspectives possibles liées à ceux-ci.

2

Scénarios de compositions fonctionnelles

Sommaire

2.1 Applications de l'étude de cas 20
2.1.1 Application : *Business Application* 20
2.1.2 Application : *SocialInsurance Application* 23
2.2 Composition au niveau des IHM et problèmes sous-jacents 24
2.2.1 Juxtaposition de deux IHM 24
2.2.2 Prise en compte du flot de données au sein des IHM 27
2.2.3 Conclusion sur la composition d'IHM 27
2.3 Compositions fonctionnelles : réponse aux ambiguïtés de la composition d'IHM 28
2.3.1 Composition en parallèle de deux opérations avec sélection des sorties . . . 29
2.3.2 Composition en séquence avec utilisation du résultat 31
2.3.3 Composition en parallèle de deux opérations avec fusion des sorties 34
2.3.4 Conclusion sur la composition fonctionnelle 36
2.4 Conclusion 37

LES scénarios présentés dans ce chapitre ont pour but de montrer que l'utilisation de la composition fonctionnelle peut guider la composition des IHM des applications composées. Les scénarios reposent sur deux applications. L'une de ces applications permet d'obtenir des informations personnelles grâce au numéro de sécurité sociale, l'autre application permet d'obtenir des informations à caractère professionnel grâce au nom d'une personne. Elles contiennent des données similaires telles que l'adresse, le courriel mais dont la valeur diffère selon le contexte (domicile / entreprise). Ces deux applications sont composées pour créer des applications plus complexes qui répondent à des besoins spécifiques comme fournir des informations aux services d'urgence dans le cadre d'une intervention (`EmergencyApplication`). Elles sont implémentées par des *Web Services* en `Java` et des IHM en `Flex`. Par la suite, c'est sur cette implémentation que l'on illustre l'ensemble de l'approche et de la mise en œuvre.

La section 2.1 présente les applications de départ en détaillant chacune des parties qui la compose (*i.e.* la partie fonctionnelle et la partie IHM). La section 2.2 illustre les questions sur les IHM résultantes qui se posent à un développeur qui compose les applications en ne se fiant qu'aux IHM. La section 2.3 montre comment la composition fonctionnelle peut aider le développeur

face aux choix précédents pour sélectionner les éléments d'IHM à conserver. Enfin, la section 2.4 conclut ce chapitre en mettant en évidence les réponses que cette thèse a pour objectif de donner à la problématique illustrée par ces scénarios.

2.1 Applications de l'étude de cas

La description de ces deux applications sert de base aux compositions qui sont présentées ensuite. Elles reposent une description de la partie fonctionnelle indépendante de la description de l'IHM. La partie fonctionnelle qui s'appuie sur des *Web Services* a pour avantage de **distinguer la définition des opérations et la définition des données manipulées**. L'utilisation de la technologie `Flex` (MXML et *Action Script*) permet de **distinguer la description de l'IHM de l'interaction** avec la partie fonctionnelle. C'est sur ce type d'IHM que sont présentées, en section 5.1.1, les transformations d'abstractions.

La première application (cf. section 2.1.1) permet d'obtenir des informations au travers de données professionnelles telles qu'un annuaire d'entreprise, *Business Application* qui fournit des informations à caractère professionnel. La seconde application (cf. section 2.1.2) permet d'obtenir des informations au travers des données de la Sécurité Sociale : *SocialInsurance Application* fournit des informations personnelles sur l'assuré. Ces deux applications respectent une architecture de type MVC [Ree79] entre la partie fonctionnelle décrite au travers de *Web Services* en `Java`, la partie IHM décrite en MXML et les interactions qui existent entre ces deux parties décrites en *Action Script*. Pour simplifier la lecture, l'ensemble des interfaces des *Web Services* (WSDL), des schémas de données ainsi que le code des IHM se trouvent en annexe A.1 ; cette section ne contient que les principaux éléments.

2.1.1 Application : *Business Application*

L'utilisateur, un administratif hospitalier, interagit avec deux IHM qui permettent d'obtenir : soit des informations sur un employé soit les différentes adresses de ses lieux de travail. Ces informations sont obtenues en fournissant le nom complet (prénom et nom de famille) de l'employé. Ces IHM permettent également d'afficher dans le premier cas les informations de l'employé telles que : sa position, ses adresses professionnelles, bâtiments et bureaux…et dans le second cas la liste de ses adresses professionnelles. Ces informations sont fournies par le *Web Service BusinessService* en réponse à une requête faite par un utilisateur.

Ce qui suit présente des extraits du *Web Service*, des extraits du schéma de données utilisé pour les deux opérations du *Web Service*, et la présentation des IHM associées à ce *Web Service*.

Le *Web Service* : *BusinessService*

Ce service propose deux opérations : (i) `getBusinessInfo` et (ii) `getAddresses`. Ces opérations prennent en paramètre un nom complet qui correspond à un prénom et un nom de famille. L'opération `getBusinessInfo` retourne des informations concernant un employé tandis que l'opération `getAddresses` ne retourne que les adresses professionnelles de cet employé. Les informations concernant un employé sont décrites dans le schéma `BusinessService.xsd` (listing A.5) et présentées dans le listing 2.1. Cette structure de données contient les différents éléments qui composent une information relative à un employé qui sont :

- un ensemble d'adresses (`address`) représenté par un tableau de chaînes de caractères,
- un ensemble de bâtiments (`building`) représenté par un tableau de chaînes de caractères,
- son courriel (`email`) représenté par une chaîne de caractères,
- son nom complet (`fullName`) représenté par une chaîne de caractères,
- un ensemble de bureaux (`office`) représenté par un tableau de chaînes de caractères,
- sa position au sein de l'entreprise (`position`) représentée par une chaîne de caractères.

On reconnait les tableaux grâce au mot clé `unbounded`.

```
<xs:complexType name="businessInformation">
  <xs:sequence>
    <xs:element name="address" type="xs:string" nillable="true"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="building" type="xs:string" nillable="true"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="email" type="xs:string" minOccurs="0" />
    <xs:element name="fullName" type="xs:string" minOccurs="0" />
    <xs:element name="office" type="xs:string" nillable="true"
                minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="position" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Listing 2.1 – Extrait du schéma de données qui correspond à la déclaration du type complexe `businessInformation`

Le listing 2.2, extrait du WSDL, présente l'opération `getBusinessInfo`. Cette opération reçoit un message en entrée de type `getBusinessInfo` et retourne en sortie un message de type `getBusinessInfoResponse`. Le schéma de données (cf. listing 2.3) spécifie que le message en entrée ne contient qu'un élément qui est une chaîne de caractères correspondant au nom complet de l'employé et que le message de sortie correspond à une `businessInformation`.

```
<message name="getBusinessInfo">
  <part name="parameters" element="tns:getBusinessInfo"></part>
</message>
<message name="getBusinessInfoResponse">
  <part name="parameters" element="tns:getBusinessInfoResponse"></part>
</message>
...
<operation name="getBusinessInfo">
  <input message="tns:getBusinessInfo"></input>
  <output message="tns:getBusinessInfoResponse"></output>
</operation>
```

Listing 2.2 – Déclaration de l'opération `getBusinessInfo`

```
<xs:element name="getBusinessInfo" type="tns:getBusinessInfo" />
<xs:element name="getBusinessInfoResponse" type="tns:getBusinessInfoResponse" />
<xs:complexType name="getBusinessInfo">
  <xs:sequence>
    <xs:element name="fullName" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="getBusinessInfoResponse">
  <xs:sequence>
    <xs:element name="return" type="tns:businessInformation" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Listing 2.3 – Extrait du schéma de données associé à l'opération `getBusinessInfo`

Le listing 2.4 présente l'opération `getAddresses`. Cette opération reçoit un message en entrée de type `getAddresses` et retourne en sortie un message de type `getAddressesResponse`. Le schéma de données (cf. listing 2.5) spécifie que le message en entrée ne contient qu'un élément qui est une chaîne de caractères qui correspond au nom complet de l'employé et que le message de sortie correspond à un tableau de chaînes de caractères.

```
<message name="getAddresses">
  <part name="parameters" element="tns:getAddresses"></part>
</message>
<message name="getAddressesResponse">
  <part name="parameters" element="tns:getAddressesResponse"></part>
</message>
...
<operation name="getAddresses">
  <input message="tns:getAddresses"></input>
```

```
  <output message="tns:getAddressesResponse"></output>
</operation>
```

Listing 2.4 – Déclaration de l'opération `getAddresses`

```
<xs:element name="getAddresses" type="tns:getAddresses" />
<xs:element name="getAddressesResponse" type="tns:getAddressesResponse" />
<xs:complexType name="getAddresses">
  <xs:sequence>
    <xs:element name="fullName" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="getAddressesResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" nillable="true"
                minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

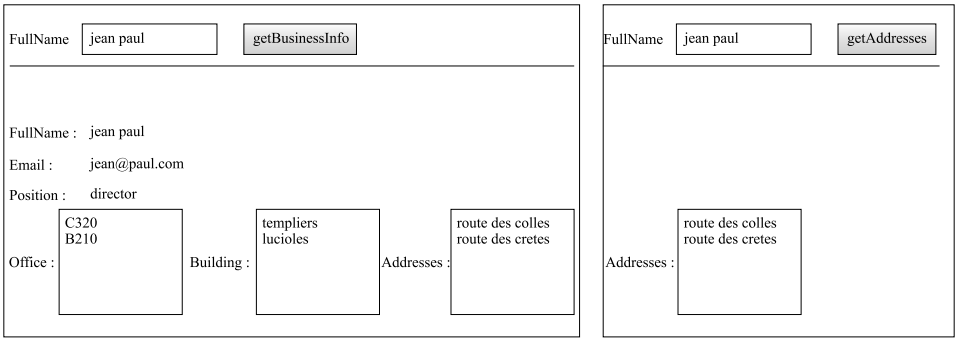
Listing 2.5 – Extrait du schéma de données associé à l'opération `getAddresses`

Pour conclure, le *Web Service BusinessService* possède deux opérations `getBusinessInfo` et `getAddresses` dont les prototypes sont les suivants :

- `businessInformation getBusinessInfo (String fullName)`
- `String[] getAddresses (String fullName)`

IHM de l'application *Business Application*

Le *Web Service* (*BusinessService*), présent dans l'application (*Business Application*) fournit deux opérations qui prennent en paramètre la même information qui est le nom complet mais qui retournent soit un ensemble de données sur l'employé, soit juste ses adresses professionnelles. Chacune de ces opérations possèdent une IHM qui permet à l'utilisateur de fournir les données nécessaires à l'exécution de l'opération et qui permet de visualiser le résultat en retour. Ces deux IHM sont illustrées dans la figure 2.1. L'IHM de la figure 2.1(a) correspond à l'opération `getBusinessInfo` et fournit un champ de saisie pour fournir le nom et prénom, un bouton pour déclencher l'appel de l'opération et six champs pour afficher les valeurs de retours (cf. listing A.6). L'IHM de la figure 2.1(b) correspond à l'opération `getAddresses` et ne possède qu'un champ de retour pour afficher les adresses.



(a) IHM de l'opération `getBusinessInfo` (après exécution)

(b) IHM de l'opération `getAddresses` (après exécution)

FIGURE 2.1 – IHM associées au *Web Service BusinessService* présentes au sein de l'application *Business Application*

2.1.2 Application : *SocialInsurance Application*

Les informations d'un assuré : nom, prénom, adresse, courriel, date de naissance, médecin référent et numéro d'assuré, sont obtenues par le *Web Service SocialInsuranceService* en réponse à une requête faite par un utilisateur d'une administration hospitalière. Ce *Web Service* possède deux opérations qui fournissent l'information associée à un assuré soit à partir de son numéro de *Carte Vitale* soit à partir de son nom et prénom.

Ce qui suit présente des extraits du *Web Service* ainsi que du schéma de données utilisé pour les deux opérations du *Web Service*. A chaque opération est associée une IHM décrite par la suite. L'intérêt de distinguer une IHM par opération est de rendre indépendantes les opérations au niveau présentation et d'ainsi laisser le choix de la manière d'obtention de l'information.

Le *Web Service* : *SocialInsuranceService*

Ce service propose de deux opérations : (i) `getByName` et (ii) `getByCard`. Ces opérations prennent en paramètre soit (i) un nom et un prénom, soit (ii) un numéro de *Carte Vitale*. Ces deux opérations retournent la même structure de données fournie par le schéma suivant `SocialInsuranceService.xsd` (cf. listing A.2). Le WSDL du *Web Service* est quant à lui présenté dans le listing A.1. On ne détaille pas ici la description de chacune des opérations. On ne détaille que la structure de données utilisées en retour des deux opérations et qui est présentée par le listing 2.6. Cette structure contient les champs composant une information relative à un assuré sociale (`socialInsuranceInformation`). Cette structure contient les éléments suivants :

- son adresse (`address`) représentée par une chaîne de caractères,
- sa date de naissance (`date`) représenté par un `dateTime` qui est une date complète (jour, mois, année, heure, minute, seconde),
- son courriel (`email`) représenté par une chaîne de caractères,
- son prénom (`firstName`) représenté par une chaîne de caractères,
- son numéro d'assuré (`insuranceNumber`) représenté par un nombre (`long`),
- le nom de famille de l'assuré (`lastName`) représenté par une chaîne de caractères,
- son médecin référent (`referee`) représenté par une chaîne de caractères.

```
<xs:complexType name="socialInsuranceInformation">
  <xs:sequence>
    <xs:element name="address" type="xs:string" minOccurs="0" />
    <xs:element name="date" type="xs:dateTime" minOccurs="0" />
    <xs:element name="email" type="xs:string" minOccurs="0" />
    <xs:element name="firstName" type="xs:string" minOccurs="0" />
    <xs:element name="insuranceNumber" type="xs:long" />
    <xs:element name="lastName" type="xs:string" minOccurs="0" />
    <xs:element name="referee" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Listing 2.6 – Extrait du schéma de données qui correspond à la déclaration du type complexe `socialInsuranceInformation`

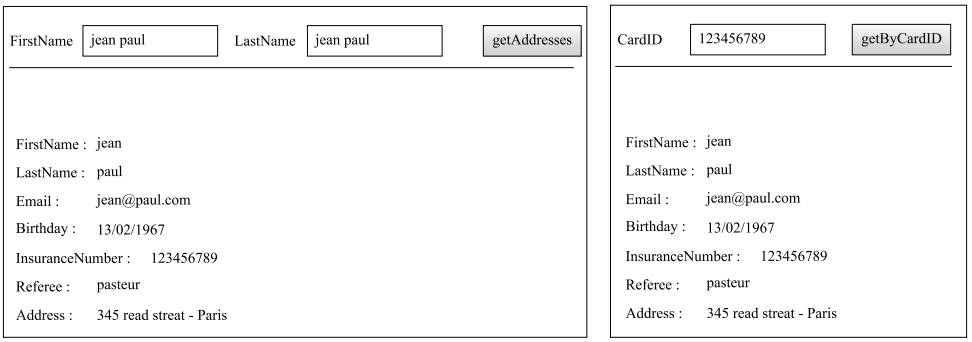
Le *Web Service SocialInsuranceService* possède deux opérations `getByCard` et `getByName` dont les prototypes sont les suivants :

- `socialInsuranceInformation getByCard(long cardID)`
- `socialInsuranceInformation getByName(String firstName, String lastName)`

Les IHM du *Web Service SocialInsuranceService*

Le *Web Service* (*SocialInsuranceService*) fournit deux opérations qui retournent le même résultat mais qui demandent des paramètres différents pour les obtenir. A chaque opération est associée une IHM. Ces deux IHM sont illustrées dans la figure 2.2. L'IHM de la figure 2.2(a) correspond à l'opération `getByName` qui, à travers deux champs de saisie fournissent le nom et le prénom, un

bouton pour déclencher l'appel de l'opération, sept champs pour afficher les valeurs de retours. L'IHM de la figure 2.2(b) correspond à l'opération `getByCard` et ne possède qu'un champ de saisie pour fournir le numéro de Carte Vitale (cf. listing A.3).



(a) IHM de l'opération `getByName` (après exécution)

(b) IHM de l'opération `getByCard` (après exécution)

FIGURE 2.2 – IHM associées au *Web Service SocialInsuranceService* présentes au sein de l'application *SocialInsurance Application*

2.2 Composition au niveau des IHM et problèmes sous-jacents

La composition d'IHM permet de réaliser des fusions d'éléments graphiques en se basant sur des **opérateurs de composition** (cf. section 3.2.1) tels que l'union, la sélection…Ces compositions d'IHM **ne considèrent pas la partie fonctionnelle** de l'application.

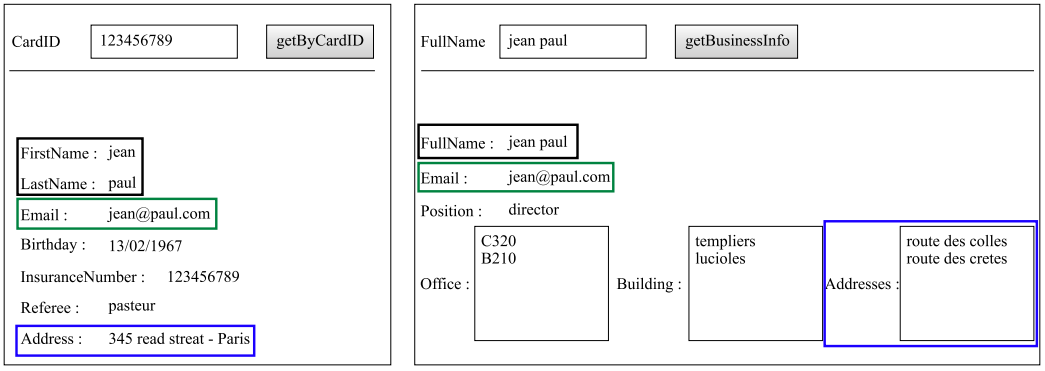
L'objectif de cette section est de présenter les inconvénients qu'il y a à ne s'intéresser qu'à la composition d'IHM sans se préoccuper de la partie fonctionnelle. Des ambiguïtés peuvent apparaître et des choix doivent être faits par le développeur qui compose. Cela nécessite donc une attention particulière afin de décider quels sont éléments qui doivent être conservés. Par ailleurs, ces choix peuvent entraîner des erreurs (*e.g.* lorsque le développeur fusionne deux éléments mais que les résultats fournis par les opérations diffèrent). Ces erreurs peuvent apparaître car les choix effectués se font en aveugle sans la connaissance de l'usage qui est fait des éléments d'IHM au niveau fonctionnel. Il n'est donc pas possible d'assurer que l'utilisation des fonctionnalités présentes dans le noyau fonctionnel sera toujours possible après composition (ce qui peut se produire si on supprime une entrée qui est nécessaire à l'exécution de l'opération).

Les compositions qui suivent illustrent, de manière non exhaustive, différentes possibilités de compositions avec les problèmes associés à chacune d'elles. La première composition illustre une fusion des deux IHM comme le suggère les travaux sur la composition de description d'IHM (cf. section 3.2.1). Cette composition ne s'appuie que sur la structure de l'IHM. La seconde composition illustre l'utilisation d'informations de l'une des IHM comme entrée de la suivante ce qui se rapproche des possibilités offertes par l'utilisation des arbres de tâches (cf. section 3.2.2). Cette seconde composition n'est réalisable que si l'on connait les données manipulées au sein des IHM.

2.2.1 Juxtaposition de deux IHM

La juxtaposition des deux IHM présentées dans les figures 2.3(a) et 2.3(b) pose un certain nombre de problèmes de redondances et de similitude d'informations dans le sens où l'on retrouve des adresses (personnelles et professionnelles) dans les deux IHM. La Figure 2.3 met en

évidence les éléments qui correspondent au même type d'information dans les deux IHM. On constate donc qu'il y a trois types d'éléments communs. Le premier, retrouvé dans le cadre vert, représente la même information présentée de la même manière dans les deux IHM (ici le courriel). Le second se situe dans le cadre bleu et correspond aux adresses. Dans un cas, on a une représentation d'une seule adresse alors que dans l'autre cas cela correspond à une liste d'adresses. Faut-il alors fusionner ces deux éléments ou bien en conserver qu'un seul ? Dans le cas où l'on fusionne les deux éléments, lequel des deux faut-il choisir ? Enfin le troisième se trouve dans le cadre noir. Ici, on trouve d'un côté une information éclatée dans deux éléments tandis que l'autre l'information ne se situe que dans un seul élément. Cependant les deux fournissent des informations relatives à l'identité de la personne à savoir son nom et prénom. Encore une fois, que faut-il faire dans ce cas, tout conserver ou pas ? Et lequel ou lesquels conserver ?



(a) IHM de l'opération `getByCard` (b) IHM de l'opération `getBusinessInfo`

FIGURE 2.3 – Mise en évidence des informations communes entre les deux IHM

Plusieurs solutions permettent de répondre à ces questions avec des résultats variés d'IHM. Ces résultats peuvent être : (i) on conserve tous les éléments présents dans les deux IHM, (ii) certains éléments sont fusionnés et (iii) tous les éléments fusionnables sont fusionnés. Ils correspondent à des besoins utilisateurs différents et à des compositions fonctionnelles sous-jacentes différentes. Voici quelques exemples de résultats possibles de composition.

La Figure 2.4 conserve tous les éléments présents dans les deux IHM. Cela signifie donc que chaque élément affiche l'information en provenance du service qui lui est rattaché. Il s'agit donc une juxtaposition des deux IHM.

Dans ce qui suit (cf. figure 2.5, 2.6 et 2.7), les IHM sont fusionnées. Ainsi, on ne conserve qu'un seul bouton afin de permettre d'appeler les deux opérations des deux *Web Services* en même temps. La Figure 2.5 fusionne le courriel ; on obtient une IHM avec un élément en moins par rapport aux IHM d'origine. L'inconvénient de cette fusion réside dans le fait qu'il est nécessaire que les informations fournies par les deux opérations soient **identiques**. Sinon, cette fusion par union des éléments graphiques n'a pas de sens. Cela signifie donc que l'utilisateur doit fournir les informations concernant la même personne en entrée sous peine d'avoir des résultats incohérents.

La Figure 2.6 considère que le prénom et le nom en résultat pour l'IHM *getByCard* ne sont pas nécessaires et que seul suffit l'élément en sortie de l'opération `getBusinessInfo` qui affiche le nom complet. Cette sélection impose comme dans le cas précédent, d'être sûr que les deux opérations fournissent le même résultat (que ce soit de manière disjointe ou combinée). Si ce n'est pas le cas, le résultat fourni à l'utilisateur est erroné.

La Figure 2.7 illustre une union sans redondance où seul le nom complet est conservé. On obtient un seul courriel et les adresses personnelles et professionnelles se retrouvent au sein d'une même liste. Cette composition permet d'avoir une représentation synthétique des retours des

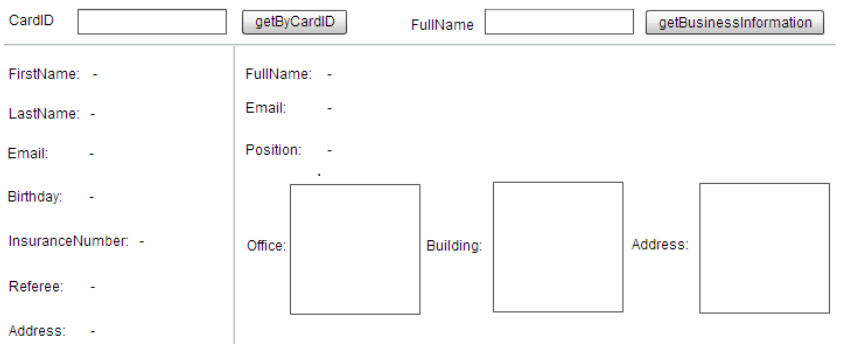


FIGURE 2.4 – Juxtaposition des fonctionnalités impliquant une redondance d'information

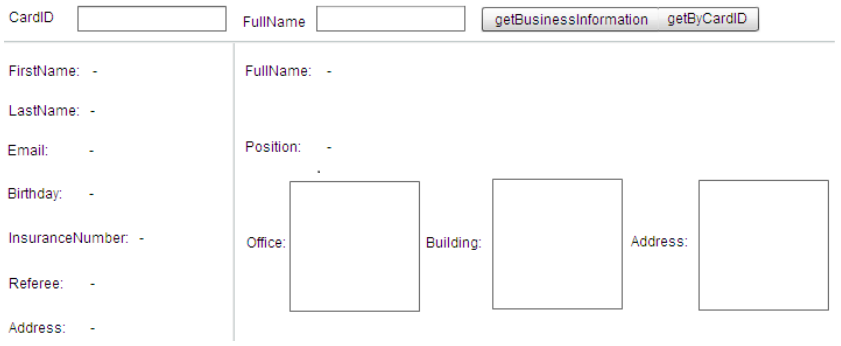


FIGURE 2.5 – Résolution de la redondance d'information par union des courriels *Email*

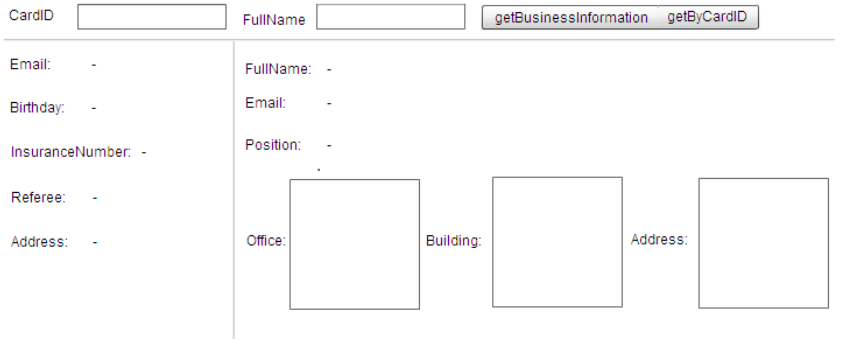


FIGURE 2.6 – Résolution de redondance d'information par sélection (conservation du nom complet *fullName* et abandon du nom et prénom)

deux opérations. Comme auparavant, cette composition n'a du sens que si les éléments pour afficher le nom/prénom et le courriel sont bien identiques.

Les résultats de composition d'IHM présentés ici s'appuient sur des opérateurs de composition que sont la sélection (qui permet de déterminer quels éléments on souhaite conserver), l'union et l'union sans redondance. Pour juxtaposer les deux IHM, il est déjà nécessaire d'effectuer une union de celles-ci (cf. section 3.2.1). La sélection s'appuie ensuite sur ce résultat (cf.

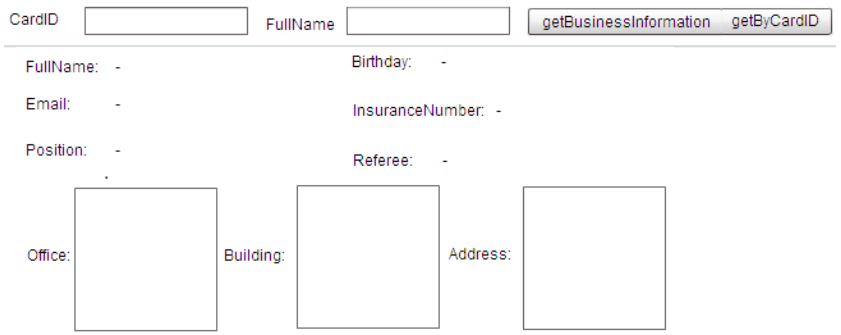


FIGURE 2.7 – Résolution de redondance d'information par union sans redondance

section 3.2.1).

2.2.2 Prise en compte du flot de données au sein des IHM

Cet exemple illustre la volonté d'utiliser les informations présentes dans une des IHM comme entrée dans une seconde IHM. La Figure 2.8 illustre le fait qu'il est nécessaire d'effectuer l'union de deux valeurs, et donc du contenu de deux éléments d'IHM, pour les fournir comme entrée.

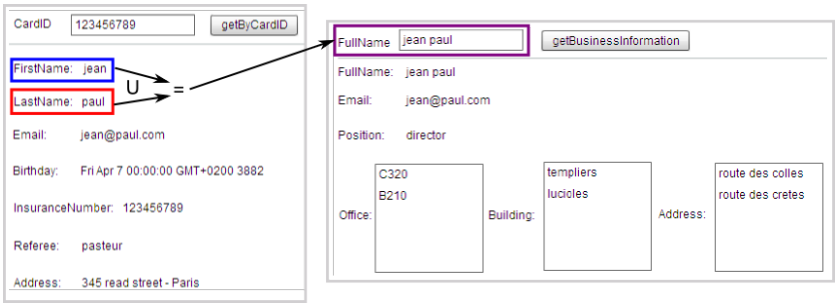


FIGURE 2.8 – Utilisation de deux sorties comme entrée dans une autre IHM

Le résultat idéal de cette composition est de n'avoir qu'un seul champ où l'on fournit le numéro de carte vitale comme entrée. Les informations que l'on récupère ainsi dans la première IHM sont fournies en entrée de la seconde afin de pouvoir appeler l'opération permettant d'obtenir les informations professionnelles concernant une personne. La Figure 2.9 illustre le résultat de cette composition. Néanmoins pour obtenir un tel résultat, il est nécessaire que les deux opérations soient enchaînées et non pas appelées en parallèle. Par conséquent, cela signifie que la composition d'IHM influence la partie fonctionnelle.

2.2.3 Conclusion sur la composition d'IHM

La composition d'IHM permet de supprimer des éléments graphiques redondants en se basant sur des **opérateurs de composition** tels que l'union, la sélection... Par ailleurs, ces compositions ne sont réalisables que sur des descriptions structurelles d'IHM homogènes qui permettent d'identifier l'ensemble des éléments graphiques et de réaliser des correspondances entre éléments graphiques (cf. section 3.2.1).

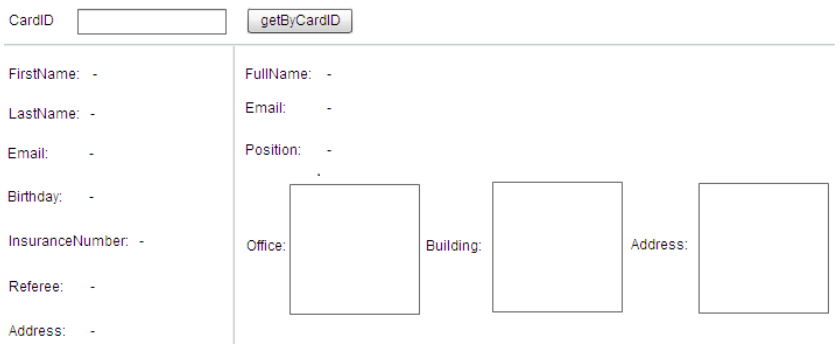


FIGURE 2.9 – Suppression d'un des champs de saisi qui est remplacé par l'utilisation des valeurs contenues dans nom (*LastName*) et prénom (*FirstName*)

Certains choix du développeur peuvent conduire à des erreurs ou à des résultats non souhaités. En effet, le développeur peut considérer que deux éléments sont identiques alors que les données affichées sont différentes. C'est ce que l'on a avec les adresses qui toutes deux affichent, d'un point vue sémantique, des adresses mais celles-ci ne sont pas identiques. Il est donc nécessaire que le développeur ait une bonne connaissance des IHM qu'ils composent afin d'effectuer les choix qui correspondent à ses attentes. L'utilisation d'éléments en sortie comme entrée d'une IHM **présuppose que les opérations soient enchaînées** et non pas exécutées en parallèle ce qui ne peut être décrit d'un point de vue IHM.

Ces ambiguïtés et erreurs que l'on peut voir dans la composition d'IHM peuvent être évitées si l'on étudie la composition fonctionnelle. En effet, la composition fonctionnelle fournit l'ensemble des informations sur l'usage qui est fait des opérations des *Web Services* composés. Il est alors possible de savoir quelles sont les données à conserver au niveau de l'IHM et les enchaînements d'opérations. C'est ce que présente la section 2.3.

2.3 Compositions fonctionnelles : une réponse aux ambiguïtés de la composition d'IHM

Les compositions fonctionnelles présentées ci-après reprennent en partie les compositions qui ont été réalisées au niveau des compositions d'IHM présentées en section 2.2. Ces compositions fonctionnelles reprennent un sous-ensemble des possibilités de composition offertes par BPEL pour réaliser des orchestrations de *Web Services*. Elles présentent ainsi des exécutions d'opérations en séquence ou en parallèle, de la fusion d'éléments et de la réutilisation de sorties en entrée. Les compositions ont été réalisées par un développeur sur deux *Web Services* mais il est possible de l'étendre à plusieurs *Web Services*.

Trois compositions sont présentées avec le flot de données associé :

- $1^{ère}$ composition : exécution en parallèle de deux opérations et sélection au niveau des sorties des opérations,
- 2^{nde} composition : exécution en séquence avec utilisation du résultat de la première opération en entrée de la seconde opération et sélection des sorties,
- $3^{ème}$ composition : exécution en parallèle de deux opérations avec fusion des sorties.

De la même manière que pour les applications, le détail des interfaces des *Web Services*, schémas de données et orchestration de services (BPEL) se trouve en annexe A.2.

Pour chacune des trois compositions, on conclut en explicitant les informations qui permettraient de résoudre la composition d'IHM.

2.3.1 Composition en parallèle de deux opérations avec sélection des sorties

Cette composition a pour but de montrer qu'il est possible de sélectionner des éléments en sortie afin d'éviter de devoir afficher l'ensemble des éléments retournés par les opérations composées. Elle est définie de la manière suivante : les deux opérations `getByCard` et `getBusinessInfo` sont exécutées en parallèle. Pour permettre l'exécution de ces deux opérations, il est nécessaire de fournir en entrée deux paramètres (*CardID* et *FullName*). Le résultat obtenu suite à l'exécution des deux opérations est un ensemble de trois valeurs de retour que sont : le nom complet (*FullName*), le courriel (*Email*) et le numéro d'assurance (*InsuranceNumber*).

La figure 2.10 illustre la composition fonctionnelle réalisée. Cette première composition possède six activités et une activité composite (*Flow*). Dans les six activités présentes, deux sont nécessaires à l'utilisation de la composition : le *receive* et le *reply* car l'opération présentée par la composition fonctionnelle prend en entrée deux paramètres et retourne trois éléments. Ensuite deux activités *assign* permettent : (i) d'assigner les paramètres d'entrée aux paramètres d'entrée des opérations qui doivent être appelées dans la composition et (ii) d'assigner les retours des deux opérations aux éléments de retour de l'opération de la composition. Enfin, deux activités *invoke* permettent d'invoquer les deux opérations `getByCard` et `getBusinessInfo`.

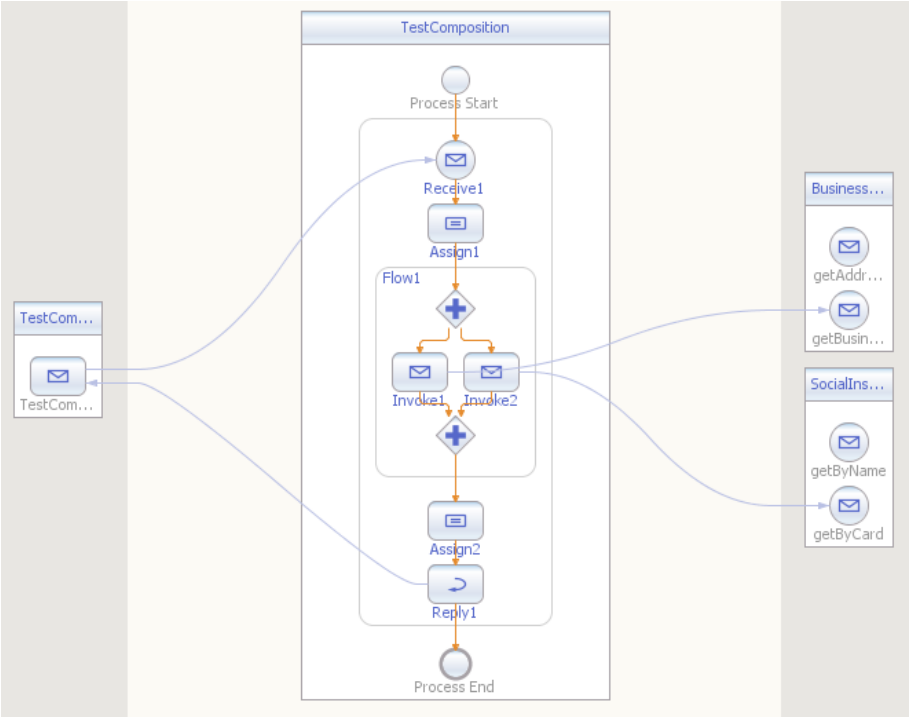


FIGURE 2.10 – *Workflow* de la première composition. On peut remarquer l'exécution en parallèle représentée par le *Flow*

La figure 2.11 illustre l'affectation qui est faite des paramètres d'entrée de l'opération de composition. Ces paramètres sont redistribués comme paramètres d'entrée des deux opérations impliquées dans la composition. Ainsi, on peut voir que *cardID* de l'opération de la composition est assigné comme paramètre d'entrée à l'opération `getByBard` et que *fullName* est assigné comme paramètre d'entrée à l'opération `getBusinessInfo`.

La figure 2.12 illustre l'affectation des valeurs de retours des deux opérations `getByCard` et `getBusinessInfo`. Ainsi, on constate que seul le numéro d'assuré (*insuraceNumber*) est conservé

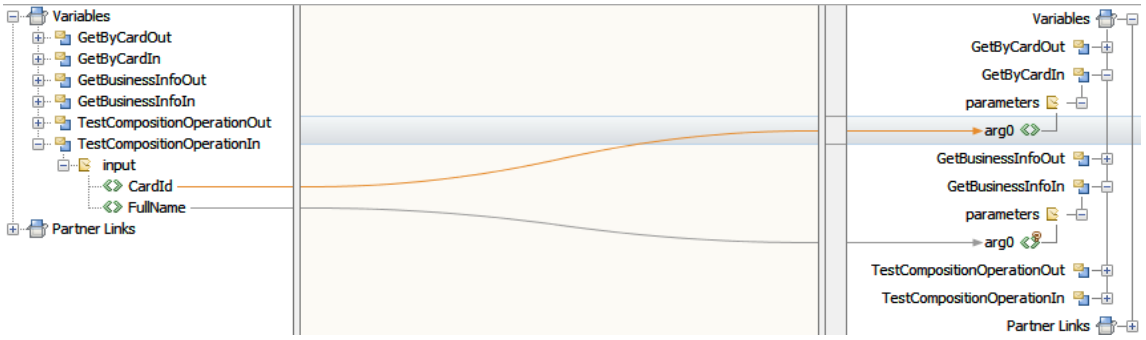


FIGURE 2.11 – Affectation des paramètres d'entrée dans une activité *assign*

de l'opération `getByCard` tandis que l'on conserve, de l'opération `getBusinessInfo`, le courriel et le nom complet (*email* et *fullName*). Ceux-ci se retrouvent comme résultats de la composition.

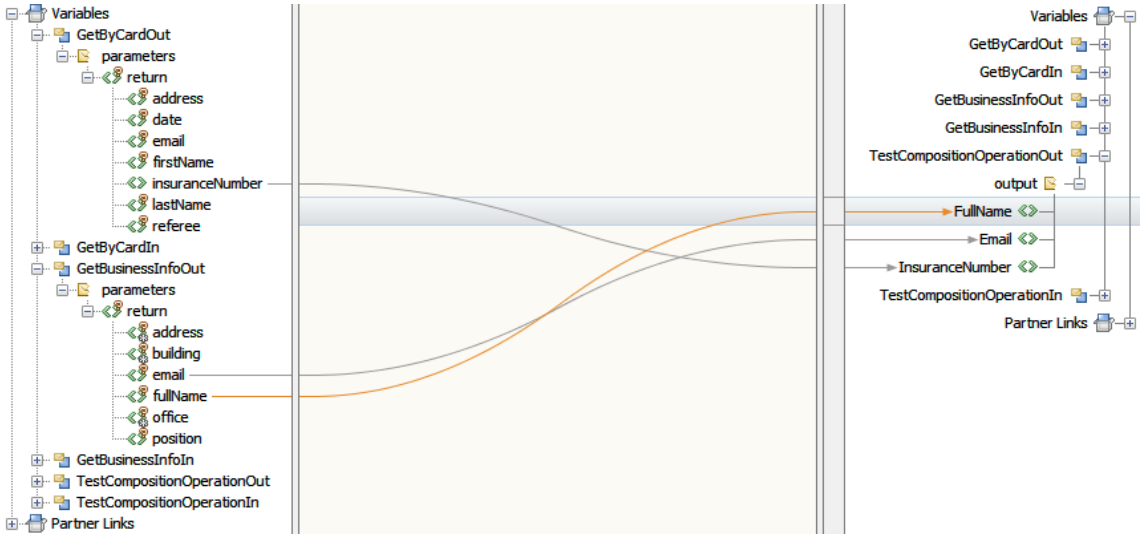


FIGURE 2.12 – Affectation des paramètres de sortie dans une activité *assign*

Une composition en parallèle avec sélection des sorties permet de :
- déterminer les éléments d'IHM en entrée pour assurer l'exécution des opérations,
- déterminer les éléments d'IHM qui déclenchent l'exécution des opérations,
- déterminer les éléments d'IHM pour rendre le résultat de l'exécution des opérations.

Ceci n'est réalisable que s'il est possible de distinguer les éléments d'IHM qui sont associés à chacune des entrées et sorties ainsi que les éléments déclencheurs des opérations composées. Il est donc nécessaire de pouvoir identifier les entrées, sorties et déclencheur d'appel d'opération au niveau des IHM des applications que l'on compose. Les compositions d'IHM présentées auparavant ne distinguent pas la notion d'entrée, sortie et déclencheur ; elles ne s'intéressent qu'au type de *widgets* que l'on compose. La composition fonctionnelle présentée ici s'apparente à la composition d'IHM présentée en figure 2.5 avec, en plus, une sélection des

éléments en sortie.

L'arbre de tâches sous-jacent correspond en la saisie par l'utilisateur des deux entrées nécessaires à l'exécution des opérations, à la réalisation des deux opérations, puis au rendu du résultat de l'exécution.

2.3.2 Composition en séquence de deux opérations avec utilisation du résultat de la première opération en entrée de la seconde

Cette composition a pour but d'exécuter en séquence les deux opérations `getByCard` et `getBusinessInfo` et d'utiliser le résultat de la première opération comme paramètre d'entrée de la seconde. Ainsi, contrairement à la première composition, il n'est plus nécessaire de fournir le nom complet (*fullName*) puisque le résultat de l'opération `getByCard` fournit le prénom et le nom de famille (*firstName* et *lastName*). Il suffit donc de coupler ces deux sorties afin d'obtenir l'entrée.

La figure 2.13 illustre la seconde composition fonctionnelle où toutes les activités sont réalisées en séquence. Cette composition possède sept activités. Comme pour la première, on retrouve les activités *receive*, *reply* et *invoke*. Encore une fois, les opérations appelées ici sont les mêmes que les opérations appelées lors de la première composition. L'opération proposée par la composition fonctionnelle prend en entrée un unique paramètre : le numéro de carte vitale (*CardID*) et retourne quatre éléments : le nom complet, le courriel, la date de naissance et l'adresse personnelle (*FullName*, *Email*, *BirthDay* et *PersonalAddress*). Les activités *assign* permettent de gérer les différentes affectations de paramètres. L'une d'entre elle va également permettre de gérer la fusion de deux éléments de sortie pour former l'entrée de la seconde opération à appeler. Dans cette composition, la première opération appelée est `getByCard` et la seconde est `getBusinessInfo`.

La figure 2.14 illustre l'affectation qui est faite pour le paramètre d'entrée de la première opération qui est exécutée. Ce paramètre provient de l'opération proposée par la composition fonctionnelle. Ainsi, le paramètre *CarId* est fourni en entrée de l'opération `getByCard`.

La figure 2.15 illustre le choix du développeur concernant le calcul du paramètre d'entrée de l'opération `getBusinessInfo` à partir de la sortie de l'opération *getByCard*. On utilise ici le prénom et le nom de famille (*firstName* et *lastName*) pour former le nom complet (*fullName*). On ajoute également un espace entre les deux afin que ceux-ci ne soient pas collés. Une fois le paramètre calculé, il est assigné au paramètre d'entrée.

La figure 2.16 illustre l'affectation des valeurs de retours des deux opérations `getByCard` et `getBusinessInfo`. Ainsi, on remarque que seul le nom complet et le courriel (*fullName* et *email*) de l'opération `getBusinessInfo` sont conservés ; la date de naissance et l'adresse personnelle (*birthDay* et *personalAddress*) sont quant à eux conservés de l'opération `getByCard`. Ceux-ci se retrouvent comme résultat de l'opération de composition.

Cette composition fonctionnelle illustre la possibilité d'utiliser des paramètres de sorties comme paramètre d'entrée d'une autre opération. Cette composition correspond à l'idée de la composition d'IHM présentée en section 2.2.2.

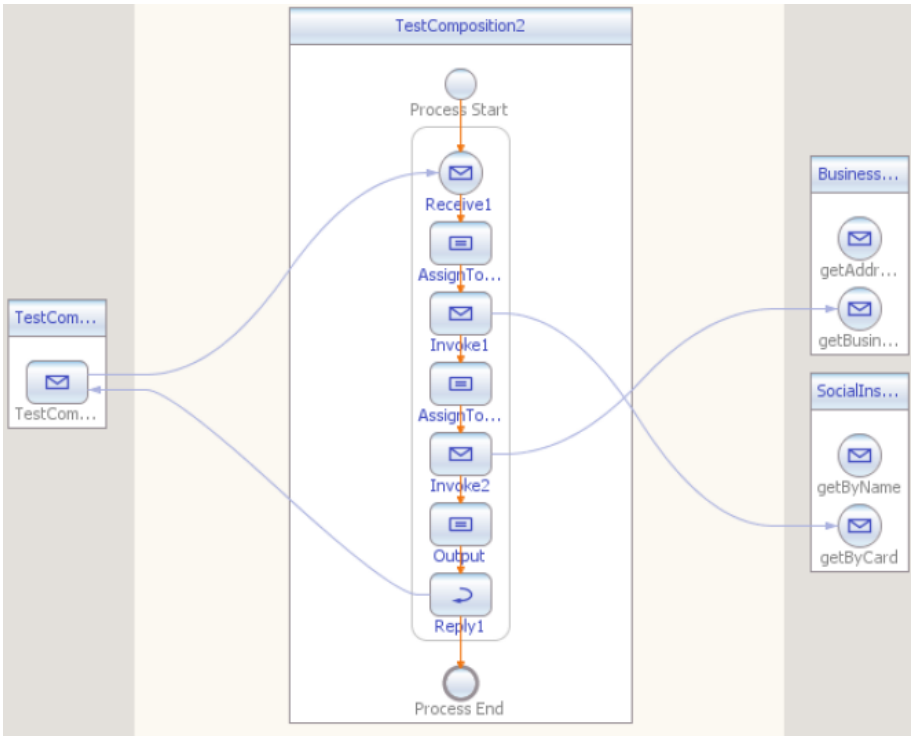


FIGURE 2.13 – *Workflow* de la seconde composition. Ici toutes les activités sont exécutées en séquence

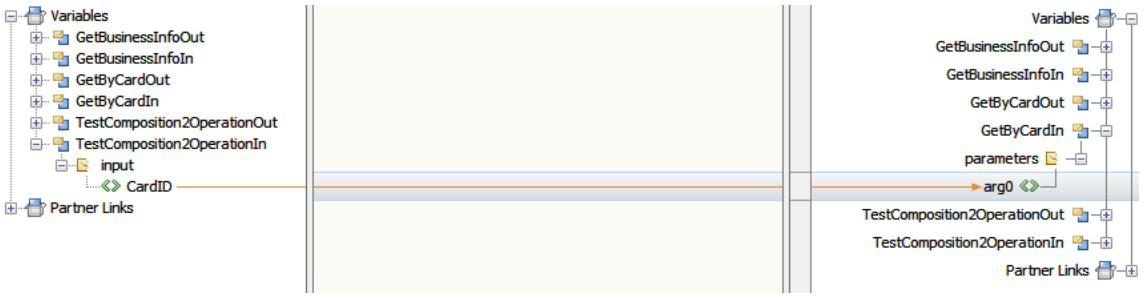


FIGURE 2.14 – Affectation du paramètre d'entrée pour l'opération *getByCard*

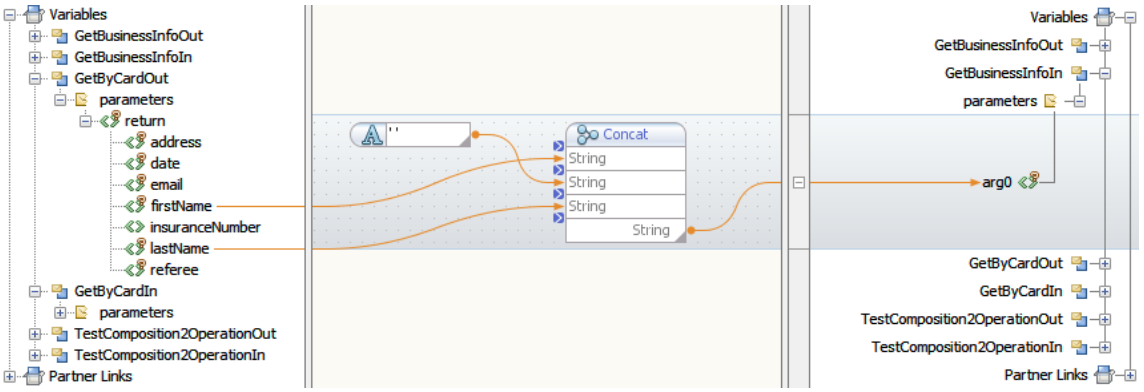


FIGURE 2.15 – Calcule du paramètre d'entrée de l'opération *getBusinessInfo* à partir des éléments de sorties de l'opération *getByCard*

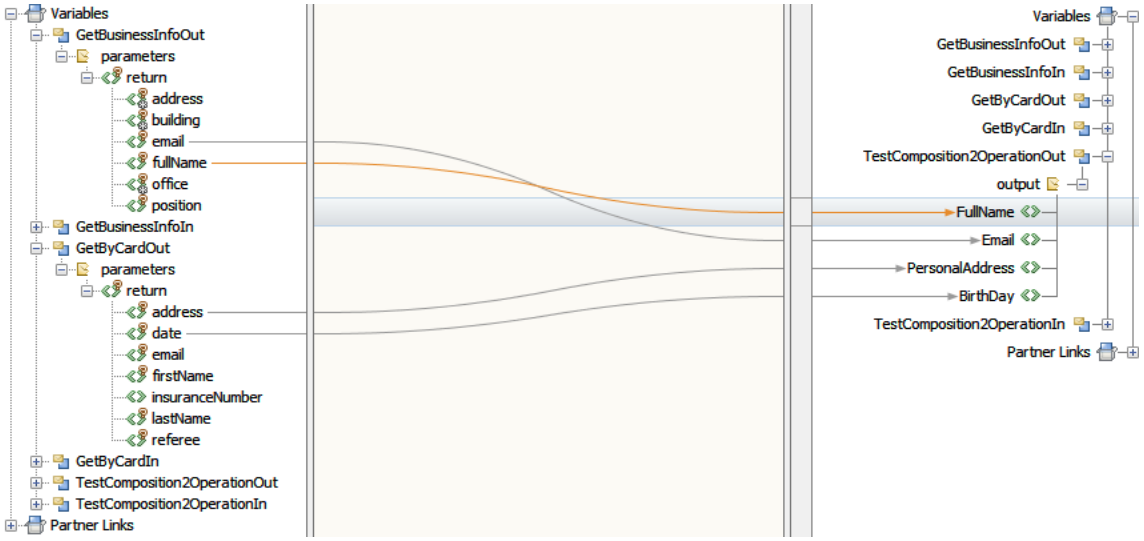


FIGURE 2.16 – Affectation des paramètres de sortie où certains sont sélectionnés

Une composition en séquence avec utilisation de sorties d'une opération en entrée d'une autre implique de pouvoir identifier comme dans la première composition fonctionnelle l'ensemble des éléments d'IHM d'entrée, de sortie et qui permettent de déclencher l'appel d'opération.

Ainsi, de cette composition, on peut réutiliser des éléments de sortie en entrée et l'on connaît les éléments d'IHM à conserver. L'intérêt d'utiliser des sorties en entrée d'autres opérations est de pouvoir éviter des erreurs de saisie de la part de l'utilisateur. Cette composition s'apparente à la composition d'IHM présentée en section 2.2.2 avec une sélection des éléments en sortie.

De la même manière que dans la composition précédente, il est nécessaire, pour identifier les éléments d'IHM à conserver, de connaître les relations qu'il y a entre l'IHM et le fonctionnel.

2.3.3 Composition en parallèle de deux opérations avec fusion des sorties

La dernière composition illustre une fusion réalisée au niveau des sorties des deux opérations composées. L'exécution se déroule, en parallèle, comme dans le premier cas. Les opérations impliquées dans cette composition sont : `getByCard` et `getAddresses`. L'objectif est d'ajouter au tableau d'adresses professionnelles retourné par l'opération `getAddresses`, l'adresse personnelle présente dans les informations obtenues par l'opération `getByCard`. Cette composition prend deux paramètres en entrée : un numéro de carte vitale (*CardId*) et un nom complet (*FullName*) et retourne un unique paramètre qui est un tableau d'adresses (*Addresses*) de type tableau de chaînes de caractères).

La figure 2.17 illustre la composition fonctionnelle qui est réalisée. Cette composition, comme la première, possède les mêmes activités. La différence réside au niveau d'un des *invoke* où s'effectue l'appel de l'opération `getAddresses` au lieu de `getBusinessInfo` et au niveau des *assign* afin de créer le bon tableau de retour pour la sortie de l'opération proposée par la composition.

La figure 2.18 illustre l'affectation des paramètres d'entrée des deux opérations qui sont appelées. Ainsi le paramètre *CardId* est passé en paramètre d'entrée de l'opération `getByCard` et le paramètre *FullName* est passé en paramètre d'entrée de l'opération `getAddresses`.

La figure 2.19 illustre l'affectation qui est faite au niveau de la valeur de retour de la composition. Pour ajouter l'adresse personnelle à la fin du tableau d'adresse professionnelle, il faut calculer le nombre d'éléments présents dans le premier tableau. Ceci est réalisé par la fonction *Count* qui est propre à la composition. On obtient donc, à la fin, un tableau qui contient toutes les adresses.

Cette composition fonctionnelle illustre les possibilités de pouvoir agréger les valeurs de retour qui proviennent de deux opérations pour n'obtenir qu'une seule valeur de retour. Dans cet exemple, cela passe par l'ajout d'un élément (l'adresse personnelle) dans une liste constituée d'un ensemble d'éléments (l'ensemble des adresses professionnelles).

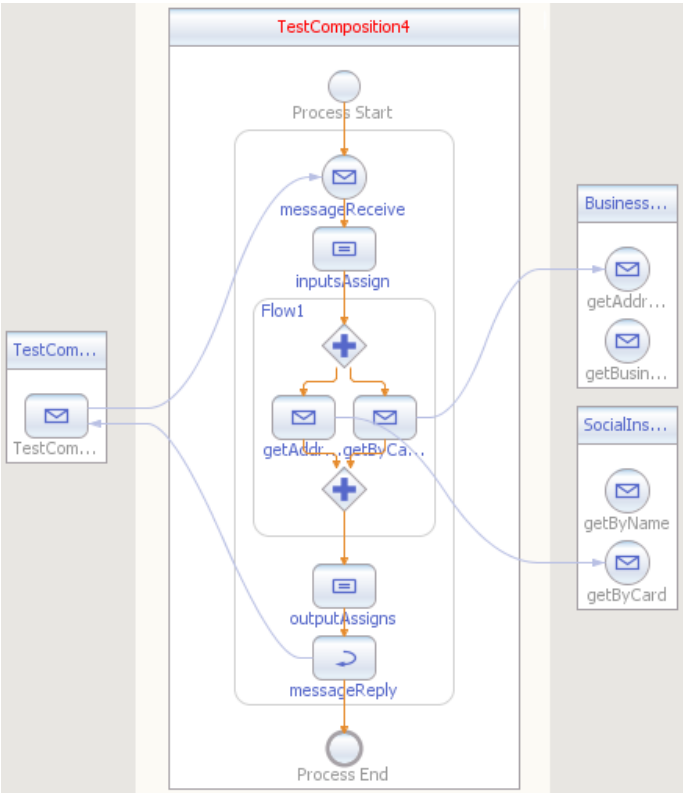


FIGURE 2.17 – *Workflow* de la troisième composition. Comme pour la première composition, les opérations sont exécutées en parallèle

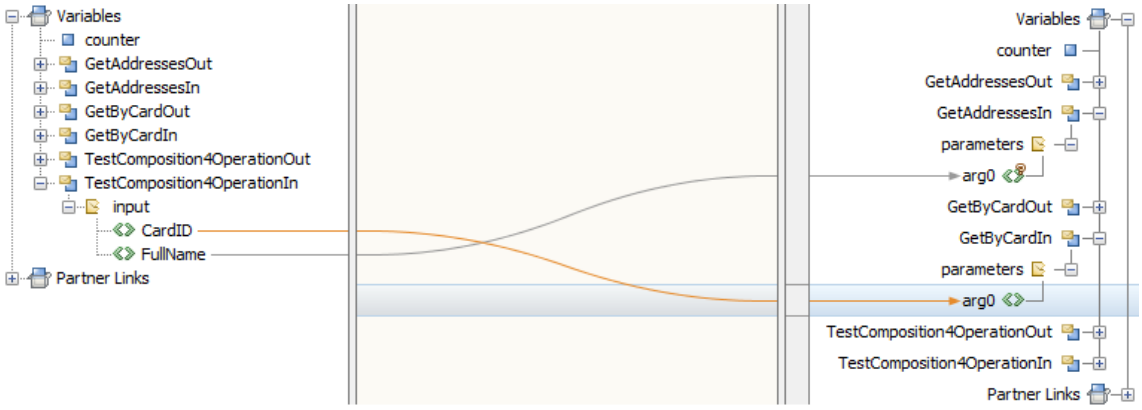


FIGURE 2.18 – Affectation des paramètres d'entrée

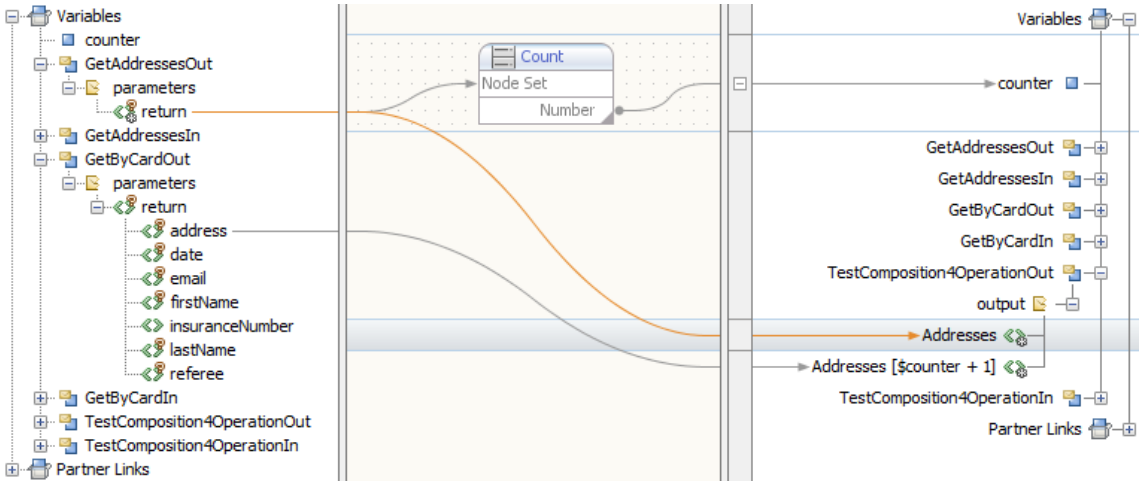


FIGURE 2.19 – Affectation du paramètre de retour avec fusion des paramètres de sorties des deux opérations

Une composition en parallèle avec fusion de sorties implique les mêmes contraintes que dans la première composition fonctionnelle présentée. Mais cela implique également d'être capable d'identifier, par la suite, les sorties au niveau des IHM ce qui revient à identifier les éléments d'IHM et si ceux-ci sont fusionnables. Cette composition s'apparente à la composition d'IHM présentée en figure 2.7 avec une sélection au niveau des sorties en plus.

L'arbre de tâches sous-jacent correspond en la saisie des entrées nécessaires à l'exécution par l'utilisateur, à l'exécution des deux opérations par le système et enfin en l'affichage du résultat.

La sélection des éléments d'IHM pour afficher le résultat devient plus complexe puisqu'il faut déterminer quel élément permet cet affichage, et dans le cas où aucun élément ne le permet, d'en créer un nouveau. Il ne peut s'agir d'un choix immédiat comme dans les autres cas. Encore une fois, ceci n'est possible que s'il est possible d'identifier pour chaque entrée et sortie du fonctionnel l'élément d'IHM correspondant.

2.3.4 Conclusion sur la composition fonctionnelle

La composition fonctionnelle fournit un ensemble d'informations sur l'usage qui est fait des données et des opérations. Il est ainsi possible, en ce qui concerne les données, de fusionner des éléments en entrée (une donnée utilisée plusieurs fois), de fusionner des éléments en sortie (agrégation de plusieurs données pour n'en avoir plus qu'une), de sélectionner des sorties, d'utiliser un résultat en sortie comme entrée d'une autre opération. En ce qui concerne les appels d'opérations, de pouvoir les effectuer en séquence, en parallèle, de les conditionner...

L'étude de la composition fonctionnelle permet de faire ressortir les éléments nécessaires à l'exécution de celle-ci ainsi que les éléments attendus en retour. Ceci permet de **déterminer quels sont les éléments d'IHM qui doivent alors être conservés ou parmi lesquels un choix doit être fait**.

Cela démontre également la nécessité pour les applications de suivre **une architecture particulière avec une bonne séparation des préoccupations afin de pouvoir identifier la partie fonctionnelle de la partie IHM et que pour chaque élément du fonctionnel il soit possible de lui**

faire correspondre un élément d'IHM. Dans le cas contraire, la composition fonctionnelle seule peut servir de base à la génération d'une nouvelle IHM mais pas en la réutilisation des IHM existantes.

2.4 Conclusion

Les scénarios de ce chapitre ont permis d'illustrer par l'exemple que les informations contenues dans la composition fonctionnelle permettent d'identifier clairement quelles sont les opérations fournies et pour chacune d'elles quelles sont les entrées et les sorties requises de l'application construite par composition. Ces informations associées à une architecture logicielle qui sépare noyau fonctionnel, IHM et interactions entre les deux comme préconisé dans MVC [Ree79], PAC [Cou87] ou Arch [BC92], rendent possible l'identification des éléments d'IHM nécessaires à l'application résultante. Une fois ces éléments déterminés il devient plus simple d'appliquer une composition d'IHM pour obtenir une application utilisable et fonctionnelle.

Le travail de thèse s'attache donc à déterminer comment extraire les informations utiles des applications existantes et des compositions fonctionnelles mises en place afin de réutiliser les IHM existantes et les composer. Suite à cette étude de cas, le chapitre suivant présente un état de l'art axé sur le domaine de la composition selon les trois axes suivants : composition fonctionnelle (cf. section 3.1), composition d'IHM (cf. section 3.2) et composition d'applications (cf. section 3.3) afin de pouvoir clairement identifier les travaux qui pourront être exploités et complétés pour atteindre l'objectif : une composition d'applications s'adressant à des développeurs logiciels soucieux de la réutilisation de l'existant.

3

Domaine d'étude : la composition d'applications

Sommaire

3.1 La composition au niveau fonctionnel 40
3.1.1 La composition dans les approches orientées services 40
3.1.2 La composition dans les approches orientées composants 47
3.1.3 Synthèse de la composition fonctionnelle 56
3.2 La composition au niveau des IHM 56
3.2.1 La composition de description d'IHM 57
3.2.2 La composition d'arbres de tâches 64
3.2.3 Synthèse de la composition d'IHM 68
3.3 La composition d'applications 68
3.3.1 Compositions dirigées par l'IHM 68
3.3.2 Compositions dirigées par le fonctionnel 74
3.3.3 Synthèse sur la composition d'applications 77
3.4 Synthèse de l'état de l'art 78

Ce chapitre présente l'état de l'art sur la composition dans le domaine des IHM et du NF afin d'identifier les points forts et les limites des travaux actuels pour atteindre la composition d'applications dans leur intégralité. L'état de l'art se découpe en trois parties : bilan des travaux sur la composition des IHM (cf. section 3.2), sur la composition des applications logicielles (cf. section 3.1) et sur la composition d'applications interactives *i.e.* comprenant une IHM (cf. 3.3). Dans l'étude de cas nous avons montré la nécessité pour les applications à composer de s'appuyer sur une architecture de type MVC [Ree79], PAC [Cou87] ou Arch [BC92] qui permet de distinguer l'IHM, le noyau fonctionnel et l'interaction entre les deux. Aussi, avons-nous ciblé dans la suite uniquement des approches qui respectent un tel découpage.

La section 3.1 présente différentes techniques de composition au niveau fonctionnel dans un but d'identifier l'utilisation qui est faite des données et des opérations et d'identifier les éléments requis (*i.e.* les paramètres d'entrée et de sorties de chaque opération).

La section 3.2 présente les solutions adoptées pour composer des IHM. Ces compositions s'appuient sur des descriptions d'IHM à un niveau abstrait permettant d'identifier les éléments qui sont présents et d'établir des correspondances entre deux éléments d'IHM afin de pouvoir les fusionner. Elles utilisent pour certaines la description des tâches utilisateurs (cf. section 3.2.2) qui s'apparentent au *workflow* présent au niveau fonctionnel (cf. section 3.1.1) mais se placent en amont dans le cycle de développement et nécessitent une génération des IHM.

La section 3.3 présente les travaux actuels traitant de la composition d'applications interactives suivant un découpage respectant les parties qui les composent (fonctionnelle, IHM et interaction). Ces compositions sont soit dirigées par l'IHM (cf. section 3.3.1) soit par le noyau fonctionnel (cf. section 3.3.2).

Enfin la section 3.4 propose une synthèse générale de l'état de l'art.

3.1 La composition au niveau fonctionnel

Cette section présente des travaux qui permettent de réaliser la composition fonctionnelle. L'objectif est de montrer les apports de cette composition pour la réalisation de la composition d'IHM. En effet, ces compositions fournissent un ensemble d'informations sur l'usage qui est fait des données et des fonctions des éléments composés. Deux approches prédominent : les approches à services qui peuvent être mises en œuvre par des *Web Services* et les approches à composants.

L'ensemble de ces approches est présenté sur l'exemple décrit dans le chapitre 2.3.1 et plus particulièrement, sur la première composition fonctionnelle qui a pour objectif d'exécuter une opération de chacun des services en parallèle et de ne conserver qu'une sous-partie de l'ensemble des informations retournées. Cet exemple se base sur deux services : l'un pour obtenir des informations d'un assuré social (*SocialInsuranceService*) et l'autre pour obtenir des informations professionnelles (*BusinessService*). Ces services sont également décrits sous forme de composants.

Ce qui suit présente, tout d'abord, la composition dans les approches orientées services (cf. sous-section 3.1.1) puis dans les approches orientées composants (cf. sous-section 3.1.2). Enfin la sous-section 3.1.3 conclut par une synthèse de la composition fonctionnelle.

3.1.1 La composition dans les approches orientées services

Ce qui suit présente la définition d'un service et plus précisément la notion de *Web Services*. Ensuite, différentes approches de composition de services sont décrites au travers de la composition de *Web Services*.

Définition de services

Les architectures orientées services (ou *Service Oriented Architecture* : `SOA`) [Erl05] sont un ensemble de principes de conception utilisé durant les phases de développement et d'intégration de système. Cette architecture s'appuie sur l'utilisation de services indépendants faiblement couplés. Ces services devant être interopérables utilisent pour ce faire des standards de description et de communication. Les standards de description permettent de normaliser le descriptif des services en s'appuyant sur l'usage d'interfaces. Les standards de communication s'assurent quant à eux que les services puissent échanger des messages et ce quelque soit le langage et la plate-forme utilisés pour créer le service. [Erl05] a défini huit caractéristiques que doivent obligatoirement posséder les architectures orientées services et plus spécifiquement les services. Ces caractéristiques sont les suivantes :

– avoir un contrat standardisé. Les services adhèrent à un contrat de communication défini collectivement par des documents qui décrivent les services.

- posséder un faible couplage. Les services maintiennent des relations qui minimisent les dépendances et qui ne requièrent qu'une connaissance des autres.
- posséder une abstraction. Les services cachent la logique métier qu'ils possèdent, seule la description du contrat (des opérations qu'ils possèdent) est montrée.
- être réutilisables. La logique métier est partagée dans plusieurs services ce qui permet la réutilisation.
- être autonomes. Chaque service contrôle sa propre logique métier qu'il encapsule, c'est-à-dire, qu'il contrôle la manière de manipuler les données qu'il utilise.
- être sans état.
- avoir la possibilité d'être découvert. Les services sont complétés par des métadonnées qui permettent l'interprétation et la découverte des services.
- être composables. Les services doivent pouvoir être efficaces au sein d'une composition quelque soit sa taille et sa complexité.

Plusieurs implémentations des services existent comme : CORBA [OMG08], DDS [OMG07], ou bien encore les *Web Services*. Ce qui suit présente plus en détail les *Web Services* qui sont utilisés par la suite afin de démontrer l'approche.

[BHM+04] donne la définition d'une architecture de *Web Services* mais également d'un *Web Service*. La définition faite d'un *Web Service* est la suivante : "Un *Web Service* est un système applicatif conçu pour supporter l'interopérabilité d'interaction entre deux machines via un réseau. Il possède une interface décrite dans un format interprétable par une machine (WSDL). Les autres systèmes interagissent avec le *Web Service* en utilisant des messages SOAP. Ces messages sont basés sur la description du *Web Service* et sont transmis via le protocole HTTP. Pour permettre le transport du message, celui-ci est sérialisé dans un format XML."

Un *Web Service* fournit une description basée sur le langage WSDL. Cette description contient l'ensemble des opérations disponibles au sein du *Web Service*. Elle spécifie également le format des messages, les types de données, le protocole de transport et le format de sérialisation. Par contre, les types de données manipulées ne se trouvent pas obligatoirement au sein du WSDL. En effet, un schéma de données basé sur XSD (*XML Schema Definition*) peut être attaché au WSDL. Ce fichier contient l'ensemble des définitions des données manipulées par les opérations du *Web Service*.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
            oasis-200401-wss-wssecurity-utility-1.0.xsd"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://rainbow.i3s.unice.fr/business"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace="http://rainbow.i3s.unice.fr/business" name="BusinessService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://rainbow.i3s.unice.fr/business"
         schemaLocation="http://localhost:14318/BusinessTest/BusinessService?xsd=1">
      </xsd:import>
    </xsd:schema>
  </types>
  <message name="getBusinessInfo">
    <part name="parameters" element="tns:getBusinessInfo"></part>
  </message>
  <message name="getBusinessInfoResponse">
    <part name="parameters" element="tns:getBusinessInfoResponse"></part>
  </message>
  <portType name="business">
    <operation name="getBusinessInfo">
      <input message="tns:getBusinessInfo"></input>
      <output message="tns:getBusinessInfoResponse"></output>
    </operation>
  </portType>
  <binding name="businessPortBinding" type="tns:business">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document">
    </soap:binding>
    <operation name="getBusinessInfo">
      <soap:operation soapAction=""></soap:operation>
```

```
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
  </binding>
  <service name="BusinessService">
    <port name="businessPort" binding="tns:businessPortBinding">
      <soap:address location="http://localhost:14318/BusinessTest/BusinessService">
      </soap:address>
    </port>
  </service>
</definitions>
```

Listing 3.1 – Extrait du WSDL du service `BusinessService` pour l'opération *getBusinessInfo*

La description des données utilise un ensemble de types primitifs utilisés pour créer des types complexes. Ces derniers peuvent décrire des séquences d'éléments qui sont soit de type simple ou bien de type complexe. Ils peuvent posséder également une occurrence minimale et maximale qui définissent la cardinalité de l'élément. L'occurrence minimale permet de savoir si l'élément est requis ou pas et l'occurrence maximale si c'est un élément unaire ou pas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://rainbow.i3s.unice.fr/business"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="1.0" targetNamespace="http://rainbow.i3s.unice.fr/business">

  <xs:element name="getBusinessInfo" type="tns:getBusinessInfo" />
  <xs:element name="getBusinessInfoResponse" type="tns:getBusinessInfoResponse" />

  <xs:complexType name="getBusinessInfo">
    <xs:sequence>
      <xs:element name="fullName" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="getBusinessInfoResponse">
    <xs:sequence>
      <xs:element name="return" type="tns:businessInformation" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="businessInformation">
    <xs:sequence>
      <xs:element name="address" type="xs:string" nillable="true"
                  minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="building" type="xs:string" nillable="true"
                  minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="email" type="xs:string" minOccurs="0" />
      <xs:element name="fullName" type="xs:string" minOccurs="0" />
      <xs:element name="office" type="xs:string" nillable="true"
                  minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="position" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing 3.2 – Schéma de données pour le Web Service `BusinessService` restreint à l'opération *getBusinessInfo*

La communication entre les *Web Services* ou entre un *Web Service* et un client se fait au travers de l'utilisation d'un protocole de communication (`SOAP` [GHM+07]). L'intérêt de ce protocole de communication est d'être indépendant d'une plate-forme d'implémentation et de permettre l'interopérabilité des *Web Services*.

Le listing 3.3 correspond à un exemple de message `SOAP` pour la demande et la réponse au service `BusinessService` pour l'opération *getBusinessInfo*.

```
POST /BusinessInformationSimple.asmx HTTP/1.1
```

```
Host: localhost
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <getBusinessInfo xmlns="http://rainbow.i3s.unice.fr/business">
      <fullName>string</fullName>
    </getBusinessInfo>
  </soap12:Body>
</soap12:Envelope>

HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <getBusinessInfoResponse xmlns="http://rainbow.i3s.unice.fr/business">
      <getBusinessInformationResponse>
        <fullName>string</fullName>
        <position>string</position>
        <email>string</email>
        <building>
          <string>string</string>
          <string>string</string>
        </building>
        <office>
          <string>string</string>
          <string>string</string>
        </office>
        <address>
          <string>string</string>
          <string>string</string>
        </address>
      </getBusinessInformationResponse>
    </getBusinessInfoResponse>
  </soap12:Body>
</soap12:Envelope>
```

Listing 3.3 – Exemple de message SOAP pour la demande et la réponse de l'opération *getBusinessInfo*

L'annuaire qui permet le référencement et la localisation des *Web Services* se nomme UDDI (*Universal Description Discovery and Integration*). Il s'agit une spécification mise au point par l'OASIS (*Organization for the Advancement of Structured Information Standards*).

La description des *Web Services* peut également se faire via l'utilisation de OWL-S (*Ontology Web Language for Services*) [MBH+06]. OWL-S est une ontologie de services qui permet la découverte automatique de services, l'invocation, la composition, l'interopérabilité et le contrôle d'exécution. Dans OWL-S, on trouve trois ontologies qui sont :

- le profil de service (*service profile*) décrit ce que le service requiert de l'utilisateur et ce qu'il leur fournit. Il décrit donc ce que le service fait ;
- le modèle de service (*service model*) décrit le fonctionnement du service et son utilisation ;
- les bases du service (*service grounding*) décrivent comment accéder au service. Elles spécifient le protocole de communication ou bien encore le format des messages.

Pour permettre l'usage d'OWL-S, le WSDL est annoté par des informations sémantiques. Ainsi, l'ensemble des opérations et des éléments qui compose les messages (*part*) est annoté par des éléments d'ontologie. Par conséquent, une opération est annotée par un *process* qui peut-être atomique ou composite. Les *process* composites sont décrits dans ce qui suit lors de la présentation de la composition de services. Un *process* atomique décrit l'ensemble de ses entrées et sorties. Chacune de ces entrées et sorties réfère à des concepts.

Le listing 3.4 est un extrait de la représentation OWL-S associée au WSDL. Celui-ci présente la description qui est faite de l'opération *getBusinessInfo*.

```
<process:AtomicProcess rdf:ID="GetBusinessInfo">
    <process:hasInput rdf:resource="#FullName_In"/>
    <process:hasOutput rdf:resource="#Address_Out"/>
    <process:hasOutput rdf:resource="#Building_Out"/>
    <process:hasOutput rdf:resource="#Email_Out"/>
    <process:hasOutput rdf:resource="#FullName_Out"/>
    <process:hasOutput rdf:resource="#Office_Out"/>
    <process:hasOutput rdf:resource="#Position_Out"/>
</process:AtomicProcess>

<process:Input rdf:ID="FullName_In">
    <process:parameterType rdf:resource="&concepts;#FullName"/>
</process:Input>

<process:UnConditionalOutput rdf:ID="Address_Out">
    <process:parameterType rdf:resource="&concepts;#OfficeAddress"/>
</process:UnConditionalOutput>

<process:UnConditionalOutput rdf:ID="Building_Out">
    <process:parameterType rdf:resource="&concepts;#Building"/>
</process:UnConditionalOutput>

<process:UnConditionalOutput rdf:ID="Email_Out">
    <process:parameterType rdf:resource="&concepts;#ProfessionalEmail"/>
</process:UnConditionalOutput>

<process:UnConditionalOutput rdf:ID="FullName_Out">
    <process:parameterType rdf:resource="&concepts;#FullName"/>
</process:UnConditionalOutput>

<process:UnConditionalOutput rdf:ID="Office_Out">
    <process:parameterType rdf:resource="&concepts;#Office"/>
</process:UnConditionalOutput>

<process:UnConditionalOutput rdf:ID="Position_Out">
    <process:parameterType rdf:resource="&concepts;#JobPosition"/>
</process:UnConditionalOutput>
```

Listing 3.4 – Extrait de la description OWL-S pour l'opération *getBusinessInfo*

L'ensemble des concepts est décrit ailleurs et permet ainsi de spécifier le type de données manipulées. Dans cet exemple, l'ensemble des références qui sont faites aux concepts correspondent à : `<!--<rdfs:subClassOf rdf:resource="&xsd;#string"/> -->`. Cela signifie que les concepts manipulés correspondent à des sous-classes d'une chaîne de caractères.

L'impact au niveau du WSDL est décrit dans le listing 3.5.

```
<message name="getBusinessInfo">
  <part name="fullName" owl-s-parameter="Business:#FullName_In"/>
</message>

<message name="getBusinessInfoResponse">
  <part name="address" owl-s-parameter="Business:#Address_Out"/>
  <part name="building" owl-s-parameter="Business:#Building_Out"/>
  <part name="email" owl-s-parameter="Business:#Email_Out"/>
  <part name="fullName" owl-s-parameter="Business:#FullName_Out"/>
  <part name="office" owl-s-parameter="Business:#Office_Out"/>
  <part name="position" owl-s-parameter="Business:#Position_Out"/>
</message>
```

Listing 3.5 – Extrait du WSDL avec les annotations OWL-S pour l'opération *getBusinessInfo*

La composition de services

La composition de services décrit à la fois : (i) l'usage qui est fait des opérations présentes au sein des services composés ; cela correspond au flot de contrôle et (ii) l'usage qui est fait des

données en spécifiant les échanges entre opérations ; cela correspond au flot de données. La combinaison des ces deux flots correspond au *workflow* de la composition de services. Le résultat de cette composition est un nouveau service, qui est déployé et publié de la même manière que tout autre service, et peut donc être réutilisé dans une autre composition.

Différentes approches ont été proposées pour réaliser la composition de *Web Services* ; [MM04] fait un tour d'horizon de ces dernières. Les orchestrations décrites en WS-BPEL permettent de décrire une composition de *Web Services*. L'utilisation du web sémantique permet également de réaliser des compositions de *Web Services* qui sont proches des orchestrations proposées par WS-BPEL. Ce qui suit présente ces deux approches de la composition de *Web Services*.

Orchestration de *Web Services* avec WS-BPEL. Les orchestrations de *Web Services* ont été présentées par [KMW03] sous le nom de BPEL4WS. Celui-ci est devenu un standard OASIS et se nomme désormais WS-BPEL [WS-07]. WS-BPEL décrit un processus métier qui correspond à un besoin spécifique. Le résultat d'une orchestration de *Web Services* est un nouveau *Web Service*. Les orchestrations de *Web Services* décrivent deux notions importantes qui sont l'usage des opérations des *Web Services* impliqués dans la composition ainsi que l'usage qui est fait des données. Les orchestrations sont définies par un ensemble d'activités décrivant le flot de données au travers d'activités *assign* et le flot de contrôle au travers des autres activités (telles que la condition, les boucles, etc.).

L'usage des données est décrit au sein d'activité d'assignation (*assign*). Dans cette activité, il est possible de décrire les différentes affectations qui sont réalisées, donc d'où proviennent les données qui vont servir en entrée d'une opération par exemple, ou bien comment le résultat d'une opération peut-être utilisé en entrée d'une autre opération... Il est également possible de faire des calculs sur ces variables au sein de cette activité, de concaténer deux données pour n'en former qu'une seule, ou d'ajouter une valeur... Cette activité permet donc de savoir comment les données sont manipulées.

Les autres activités permettent de décrire le flot de contrôle et donc l'enchaînement des opérations. Il est ainsi possible de décrire des exécutions d'opérations en séquence (*sequence*) ou en parallèle (*flow*), d'avoir la notion de condition (*if (condition)...else*), de boucle (*while*, *repeatUntil* et *forEach*) et de choix (*pick*). Tout processus BPEL doit comporter au moins deux activités qui sont : le *receive* en début de processus et le *reply* en fin de processus. Un processus BPEL étant présenté comme un nouveau *Web Service*, il est nécessaire qu'il puisse recevoir une requête et retourner la réponse de l'exécution du processus. Entre ces deux activités, se trouve l'ensemble des activités qui composent le *workflow*.

Le listing 3.6 présente un extrait de l'orchestration WS-BPEL qui correspond à la composition présentée dans le chapitre 2 et qui exécute en parallèle les opérations *getBusinessInfo* et *getByCard*.

```
  <variables>
    <variable name="GetByCardOut" messageType="tns:getByCardResponse"/>
    <variable name="GetByCardIn" messageType="tns:getByCard"/>
    <variable name="GetBusinessInfoOut" messageType="tns:getBusinessInfoResponse"/>
    <variable name="GetBusinessInfoIn" messageType="tns:getBusinessInfo"/>
    <variable name="TestCompositionOperationOut" messageType="tns:TestCompositionResponse"/>
    <variable name="TestCompositionOperationIn" messageType="tns:TestCompositionRequest"/>
  </variables>
  <sequence>
    <receive name="Receive1" createInstance="yes"
       partnerLink="TestCompoPartnerLink" operation="TestCompositionOperation"
       variable="TestCompositionOperationIn"/>
    <assign name="Assign1">
      <copy>
        <from>$TestCompositionOperationIn.input/ns0:CardId</from>
        <to>$GetByCardIn.parameters/arg0</to>
      </copy>
      <copy>
        <from>$TestCompositionOperationIn.input/ns0:FullName</from>
        <to>$GetBusinessInfoIn.parameters/arg0</to>
      </copy>
    </assign>
    <flow name="Flow1">
```

```
      <invoke name="Invoke1" partnerLink="BusinessPartnerLink"
         operation="getBusinessInfo" inputVariable="GetBusinessInfoIn"
         outputVariable="GetBusinessInfoOut"/>
      <invoke name="Invoke2" partnerLink="SocialInsurancePartnerLink"
         operation="getByCard" inputVariable="GetByCardIn"
         outputVariable="GetByCardOut"/>
    </flow>
    <assign name="Assign2">
      <copy>
        <from>$GetBusinessInfoOut.parameters/return/fullName</from>
        <to>$TestCompositionOperationOut.output/ns0:FullName</to>
      </copy>
      <copy>
        <from>$GetBusinessInfoOut.parameters/return/email</from>
        <to>$TestCompositionOperationOut.output/ns0:Email</to>
      </copy>
      <copy>
        <from>$GetByCardOut.parameters/return/insuranceNumber</from>
        <to>$TestCompositionOperationOut.output/ns0:InsuranceNumber</to>
      </copy>
    </assign>
    <reply name="Reply1" partnerLink="TestCompoPartnerLink" operation="TestCompositionOperation"
       variable="TestCompositionOperationOut"/>
  </sequence>
</process>
```

Listing 3.6 – Version simplifié du BPEL avec la description du flot de données et de contrôle

Dans ce listing, on constate qu'un ensemble d'activités s'exécute en séquence mais que les deux opérations *getBusinessInfo* et *getByCard* sont exécutées en parallèle. L'ensemble de la manipulation des données est effectué au sein des activités *assign*. Ce qui permet ainsi de voir l'affectation des paramètres d'entrée et de remarquer que pour les sorties, on conserve le courriel qui est fourni par l'opération *getBusinessInfo* et non pas celui fourni par l'opération *getByCard*.

D'autres exemples de processus BPEL se trouvent dans le chapitre 2 en section 2.3. Dans ces exemples, sont visibles l'exécution en parallèle grâce à l'activité *Flow*, ou bien encore une exécution en séquence. L'invocation d'une opération se fait par l'intermédiaire d'une activité *Invoke* qui permet d'appeler une opération d'un *Web Service* impliqué dans la composition.

Composition de *Web Services* au travers de l'utilisation d'OWL-S. OWL-S possède également un modèle de processus (*process model*), une sous-classe du modèle de service, qui décrit un service en terme d'entrées, de sorties, de pré-conditions, de post-conditions et si nécessaire ses propres sous-processus. Dans ce modèle, il est possible de décrire des processus composites. OWL-S distingue trois types de processus :

- les processus atomiques (*atomic process*) qui ne possèdent pas de sous-processus ;
- les processus simples (*simple process*) qui ne sont pas invocables directement et sont utilisés comme des éléments d'abstraction. Ainsi, ils peuvent fournir une vue pour un processus atomique et un processus composite ;
- les processus composites qui contiennent un ensemble de sous-processus. Ils sont décomposables en processus composites ou non-composites.

La décomposition des processus composites peut être spécifiée en utilisant des constructions de contrôle telles que la séquence ou la conditionnelle. Les constructions de contrôles possibles sont les suivantes :

- la *sequence* qui impose un ordre. Ce contrôle est équivalent à l'activité *sequence* dans WS-BPEL.
- le *split* où les services peuvent s'exécuter de manière parallèle,
- le *split+join* complète le *split* en imposant une synchronisation sur le résultat. Ce contrôle correspond au *flow* dans WS-BPEL. En effet, le *split* n'existe pas en WS-BPEL puisque l'ensemble des exécutions d'opérations doit être achevé avant de pouvoir continuer le *workflow*.
- l'*Any-order* (les services peuvent s'exécuter dans n'importe quel ordre mais de manière séquentielle).

- le *choice* (exécution d'un seul parmi plusieurs),
- l'*If-Then-Else* qui exprime la condition et que l'on retrouve dans WS-BPEL.
- l'*iterate* (correspond à une itération). Comparable au *forEach* de WS-BPEL.
- le *Repeat-While/Repeat-Until*. Identique à ce que l'on a dans WS-BPEL avec le *while* et le *repeatUntil*.

Un processus composite comprend plusieurs actions qui doivent être exécutées. Cela peut entraîner l'envoi et la réception d'une série de message auprès de l'utilisateur afin que celui-ci fournissent les informations nécessaires à la réalisation de l'exécution.

Le listing 3.7 correspond à la composition en parallèle des deux opérations *getByName* (du service `SocialInsuranceService`) et l'opération *getBusinessInfo* (du service `BusinessService`).

```
<process:CompositeProcess rdf:ID="Emergency_Process">
  <rdfs:label> Process to retrieve all personal and professional information </rdfs:label>
  <process:composedOf>
    <process:Split-Join>
      <process:components rdf:parseType="Collection">
        <process:AtomicProcess rdf:about="#GetBusinessInfo"/>
        <process:AtomicProcess rdf:about="#GetByCard"/>
     </process:components>
    </process:Split-Join>
  </process:composedOf>
</process:CompositeProcess>
```

Listing 3.7 – Processus composite correspondant à l'exécution en parallèle des deux opérations

Le processus ne décrit pas les entrées et sorties contrairement à WS-BPEL ; il ne décrit que l'enchaînement des opérations tandis que les entrées et sorties restent spécifiques aux opérations utilisées.

Conclusion

La composition de services s'appuie sur des architectures orientées services (SOA) qui fournissent une interface pour chacun des services. Cette interface possède la description de l'ensemble des opérations disponibles ainsi que des données manipulées par ces opérations (paramètres d'entrée et de sortie). Dans le cas des *Web Services*, ces informations se retrouvent au sein du WSDL (et du schéma de données qui peut lui être attaché).

Les compositions effectuées au niveau des *Web Services* (orchestrations) donne la possibilité de : (i) sélectionner les éléments que l'on souhaite avoir (cf. étude de cas, section 2.3), (ii) partager la même donnée en entrée, (iii) fusionner des résultats (cf. étude de cas, section 2.3.3) et (iv) utiliser un résultat comme paramètre d'entrée (cf. étude de cas, section 2.3.2).

3.1.2 La composition dans les approches orientées composants

La définition d'un composant est décrite dans ce qui suit. Par la suite, différentes plates-formes sont présentées ainsi que la composition au sein de celles-ci.

Définition de composants

La programmation par composants introduit la notion d'assemblage de composants comme technique de réutilisation. Un composant logiciel [Szy02, HC01] possède deux types d'interfaces : les interfaces fournies et les interfaces requises.

Les interfaces fournies (d'entrée) par le composant définissent les fonctionnalités offertes par un composant donné en fournissant les signatures des fonctionnalités.

Les interfaces requises (de sortie) par le composant expriment ses besoins c'est-à-dire les éléments requis pour qu'il fonctionne.

C'est au travers de ces deux interfaces qu'il est possible de connecter deux composants entre eux. Une interface requise est connectée à une interface fournie.

Une des caractéristiques du composant est l'interchangeabilité. C'est-à-dire qu'un composant peut remplacer un autre composant (que ce soit à la conception ou à l'exécution) à partir du moment où le composant remplaçant fournit, au minimum, ce que fournissait le composant remplacé. A la différence des services, un composant n'a réellement d'existence qu'au sein d'un assemblage de composants alors qu'un service doit fonctionner indépendamment de toute autre service.

On retrouve la représentation des composants dans les spécifications d'UML 2 [OMG09] dans l'utilisation des diagrammes de composants. Les interfaces fournies sont représentées par des liens en forme de "boules" et les interfaces requises par des réceptacles. Les liaisons se font entre l'interface requise et l'interface fournie (le réceptacle recevant la "boule").

La figure 3.1 représente l'assemblage de composants. Ainsi, le composant `Emergency` utilise les composants `SocialInsurance` et `BusinessInfo` afin de fournir une information jointe. La figure décrit également les données manipulées par chacun des composants.

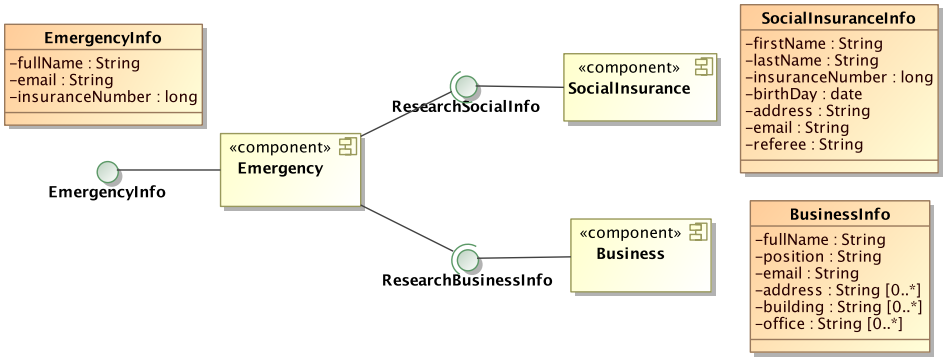


FIGURE 3.1 – Représentation de l'assemblage de composants au sein d'UML2

Les JAVABEANS [Eng97] proposent, quant à eux, d'autres types d'interface. On n'y retrouve pas les notions d'interfaces fournies et requises. En revanche, les composants JAVABEANS possèdent un ensemble de propriétés, qui sont accessibles au travers de *getter* et modifiables grâce à des *setter*, ainsi qu'un ensemble de méthodes et d'événements. C'est au travers de l'utilisation des événements qu'un composant peut notifier à d'autres composants de la modification d'une de ses propriétés. Les composants peuvent posséder des écouteurs pour réagir aux événements émis. Donc, ici, on a un ensemble de ports qui permettent d'accéder et de manipuler les données et des ports émetteurs d'événements. La notion d'événement au sein de ces composants est importante puisqu'elle est à la base du déclenchement de l'appel d'autres méthodes. C'est au travers des événements et des liens que l'on peut décrire ainsi le flot de contrôle.

Exemple de plates-formes

Il existe différentes plates-formes qui s'appuient sur l'usage de composants. Les plates-formes présentées ici sont FRACTAL, SCA et WCOMP. Ces plates-formes diffèrent par la représentation des composants, par l'usage qui en est fait ou bien par les liaisons qui sont faites. Les trois plates-formes ont des propriétés différentes : FRACTAL permet une représentation de base des composants, SCA un enrichissement du protocole de communication et une hétérogénéité des composants utilisés et WCOMP une représentation proche des *Web Services* et des possibilités de compositions offertes. Du fait de ces propriétés différentes, il est intéressant d'étudier les trois plates-formes.

La plate-forme FRACTAL. Les composants FRACTAL [BCS02, Obj08] possèdent deux types d'interfaces : les interfaces de contrôle et les interfaces fonctionnelles. Au sein des interfaces fonctionnelles, il existe également deux sous-types d'interface que sont les interfaces serveurs (interfaces fournies) et les interfaces clientes (interfaces requises).

Les interfaces de contrôle permettent la gestion des besoins non fonctionnels du composant (gestion du cycle de vie, des liaisons entre composants, etc.). Ce sont des points d'accès aux contrôleurs du composant qui permettent : l'introspection, la configuration, l'ajout de sécurité, etc.

Les interfaces fonctionnelles serveurs décrivent des méthodes fournies par le composant. Un composant peut posséder plusieurs interfaces fonctionnelles serveurs pour permettre une séparation des méthodes. Les interfaces fonctionnelles clientes décrivent ce dont le composant a besoin pour fonctionner. De la même manière que pour les interfaces fonctionnelles serveurs, un composant peut posséder plusieurs interfaces clientes. Il existe deux catégories de composants dans FRACTAL : les composants composites et les composants primitifs. Les composants primitifs sont des briques d'implémentation qui contiennent donc du code fonctionnel. Les composants composites contiennent un ensemble de sous-composants qui peuvent être des composants primitifs ou des composants composites. FRACTAL est un modèle hiérarchique de composants. La différence entre un composant composite et un composite primitif réside également dans le fait que le composant composite possède des interfaces internes qui lui permettent de se lier à ses sous-composants contrairement au composant primitif. Ainsi, l'interface fonctionnelle serveur présentée par le composant composite, devient une interface fonctionnelle cliente (pour l'interface interne) et permet ainsi de le lier à une interface fonctionnelle serveur d'un de ses sous-composants. Les interfaces de contrôles prennent une importance plus grande en ce qui concerne le composant composite puisqu'elles permettent de reconfigurer l'assemblage des sous-composants ; ainsi, à l'exécution, l'assemblage interne du composant composite peut évoluer.

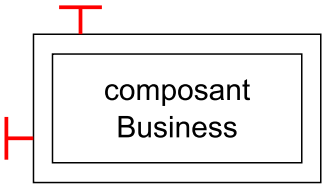


FIGURE 3.2 – Composant FRACTAL du noyau fonctionnel de l'application *Business*

La figure 3.2 illustre le composant `Business`. Ce composant est équivalent au *Web Service* précédemment décrit. La réalisation de ce composant s'appuie sur l'usage d'une interface (au sens interface `Java`). Cette interface est décrite par le listing 3.8.

```
package applications.business.fc.api;

public interface Business {
  BusinessInformation getBusinessInfo(string fullName);
  List<String> getAddresses(string fullName);
}
```

Listing 3.8 – Interface fournies par le composant `Business`

L'implémentation de cette interface va ainsi permettre de spécifier que le composant `Business` possède une interface fonctionnelle serveur qui correspond à cette interface. Le listing 3.9 illustre l'implémentation du composant `Business`. On remarque l'annotation *@Component* suivi de *provides* qui permet de spécifier que le composant fourni une interface.

```
package applications.business.fc.lib;

import org.objectweb.fractal.fraclet.annotations.Component;
import org.objectweb.fractal.fraclet.annotations.Interface;
```

```
import applications.business.fc.api.*;
import applications.dataaccess.*;

@Component(provides = {@Interface(name = "businesss",
  signature = applications.business.fc.api.Business.class)})

public class BusinessImpl implements Business {
  private DataAccess da;

  public BusinessImpl(){
    da = new DataAccess();
  }

  public BusinessInformation getBusinessInfo(string fullName) {
    return da.getBusinessInfo(fullName);
  }
  public BusinessInformation getAddresses(string fullName) {
    return da.getAddresses(fullName);
  }
}
```

Listing 3.9 – Implémentation de l'interface du composant `Business`

La plate-forme SCA. Les composants SCA [IBM06, MR09] sont similaires aux composants FRACTAL. En effet, de la même manière, on retrouve deux types d'interfaces fonctionnelles : les interfaces de types *services* correspondent aux interfaces fournies par le composant et les interfaces de types *references* correspondent aux interfaces requises par le composant. Un composant peut posséder plusieurs interfaces de type *services* et de type *references*.

Les *service*s fournissent un certain nombre d'opérations et la description de ces derniers dépend de l'implémentation utilisée. Dans le cas de composants `Java`, la description du *service* correspond à une interface `Java` et dans le cas d'un composant implémenté en BPEL, le *service* présenté correspondra à un WSDL.

Il existe deux types de composants, les composants de base et les composants composites qui regroupent d'autres composants. Le composant composite possède les mêmes caractéristiques que le composant de base, c'est-à-dire, des *service*s et des *reference*s.

Il existe cependant des différences avec les composants FRACTAL : les composants SCA peuvent être développés dans différentes technologies et un composant composite peut regrouper des composants "hétérogènes". La gestion de la communication entre les composants s'appuie sur des standards de communication tels que `SOAP`, `IIOP` (*Internet Inter-ORB Protocol*)...

Le listing 3.10 illustre la définition du composant `Business`. Ce composant est défini au travers d'une classe `Java` et utilise les annotations SCA.

```
package application.sca.business;

import org.osoa.sca.annotations.Remotable;
import applications.dataaccess.*;

@Remotable
public interface Business {
  public BusinessInformation getBusinessInfo(string fullName);
  public BusinessInformation getAddresses(string fullName);
}

public class BusinessImpl implements Business {
  private DataAccess da;

  public BusinessImpl(){
    da = new DataAccess();
  }

  public BusinessInformation getBusinessInfo(string fullName) {
    return da.getBusinessInfo(fullName);
  }
  public BusinessInformation getAddresses(string fullName) {
    return da.getAddresses(fullName);
```

```
  }
}
```

Listing 3.10 – Implémentation du composant `Business`

Ce composant est représenté par la figure 3.3 qui possède une interface *service*.

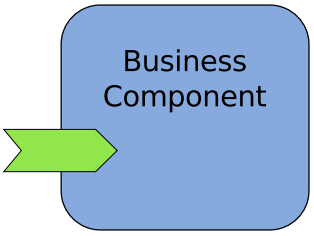


FIGURE 3.3 – Composant SCA du noyau fonctionnel de l'application *Business*

La plate-forme WCOMP. Les composants utilisés au sein de la plate-forme WCOMP sont des composants LCA (*Lightweight Component Architecture*) [TLR+09]. Ces composants dérivent des composants JAVABEANS présentés auparavant. Ainsi, ces composants possèdent deux types de ports que sont :

- les ports d'entrée, qui correspondent aux méthodes disponibles pour le composant. Ces méthodes correspondent à la logique métier du composant au même titre que pour les services mais elles permettent également de modifier des propriétés présentes au sein du composant.
- les ports de sorties qui correspondent aux événements que peut émettre le composant. Ce sont ces événements qui permettent de déclencher l'appel d'une méthode.

Le listing 3.11 correspond au composant `Business` au sein de la plate-forme WCOMP. Il possède deux attributs : le nom complet (*fullName*) et les informations professionnelles (*businessInfo*). Ces dernières ne sont accessibles qu'en lecture pour les autres composants ; il n'est pas possible de modifier le contenu. Le composant possède également un événement qui est émis lorsqu'il a récupéré les informations professionnelles.

```
using System;
using WComp.Beans;
using WComp.EventedBeans;
using Business.BusinessInfo;
using DataAcces;

namespace WComp.Basic
{
  /// <summary>
  /// Filter to write
  /// </summary>
  [Bean (Category="Basic")]
  public class Business_Bean
  {

    private DataAccess da;
    private BusinessInfo businessInfo;
    private string fullName;

    public string FullName
    {
      get
      {
        return fullName;
      }
      set
      {
        fullName = value;
```

```
      }
30    }

      public BusinessInfo getBusinessInfo()
      {
        return businessInfo;
35    }

      #region Constructor
      public Business_Bean()
      {
40      da = new DataAccess();
      }
      #endregion

      public void RetrieveBusinessInfo(string fullName)
45    {
        this.fullName = fullName;
        businessInfo = da.getBusinessInfo(fullName);
        FireRaiseInfoEvent(businessInfo);
      }
50
      #region Event
      public delegate void EventRaiseInfoHandler(BusinessInfo businessInfo);
      public event EventRaiseInfoHandler raiseInfoEvent;

55    private void FireRaiseInfoEvent(BusinessInfo businessInfo)
      {
        if(raiseInfoEvent != null)raiseInfoEvent(businessInfo);
      }
      #endregion
60  }
}
```

Listing 3.11 – Implémentation du composant `Business` dans WCOMP

L'assemblage de composants

L'assemblage des composants diffère selon la plate-forme utilisée. C'est ce que présente les paragraphes ci-dessous pour les trois plates-formes précédemment décrites.

Assemblage de composants dans FRACTAL. La création d'un assemblage de composants au sein de FRACTAL passe par l'utilisation de liaisons. Ces liaisons se font entre une interface fonctionnelle cliente et une interface fonctionnelle serveur. Pour que la connexion entre les interfaces soit possible, il est nécessaire que l'interface fonctionnelle serveur propose au moins l'ensemble des méthodes requises par l'interface fonctionnelle cliente.

Afin de présenter cet assemblage comme un nouveau composant, ceux-ci peuvent être regroupés au sein d'un composant composite. Les interfaces de contrôle présentes sur le composant composite permettent de reconfigurer celui-ci au moment de l'exécution. Cette reconfiguration peut aboutir à un changement des liaisons au sein du composant composite.

La description du composant composite, ou même des liaisons, se fait au travers de l'utilisation du FRACTAL ADL (*Architecture Description Language*). Celui-ci contient la description de l'ensemble des composants impliqués dans l'assemblage tout en spécifiant leurs interfaces clientes et serveurs. Il contient également la définition des liaisons (*binding*) qui existent entre les composants.

Dans le cas de la création d'une application fournissant les informations nécessaires aux services d'urgence, il y a la création d'un composant composite qui agrège trois composants : (i) le composant `Business`, (ii) le composant `SocialInsurance` et le composant *Emergency* qui contient la logique d'appel sur les deux autres composants. La figure 3.4 illustre cet assemblage et la création du composant composite.

Pour réaliser cet assemblage, il est nécessaire de définir une nouvelle interface qui est l'interface du composant `Emergency` (l'interface du composant `SocialInsurance` n'est pas décrite ici).

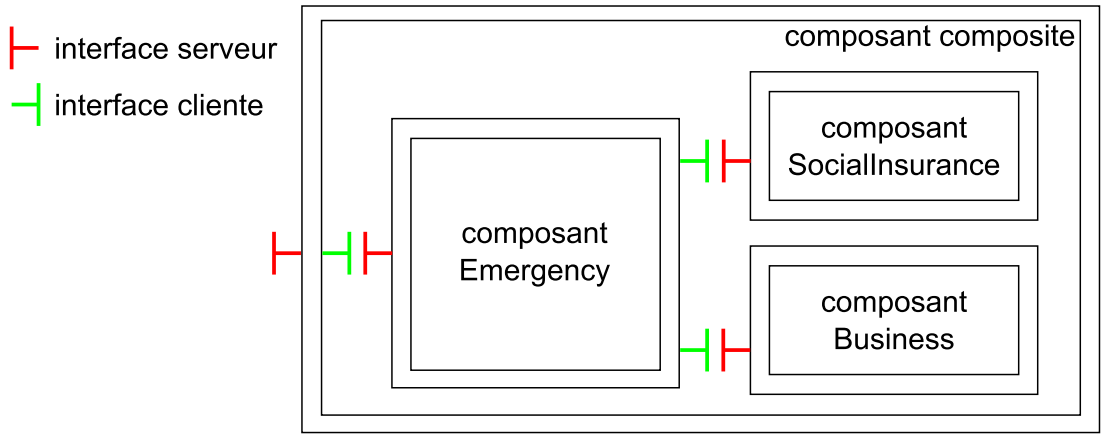


FIGURE 3.4 – Assemblage de composants FRACTAL

Le listing 3.12 correspond à la définition de cette interface.

```
package applications.emergency.fc.api;

public interface Emergency {
  EmergencyInformation getInfo(string fullName, long cardID);
}
```

Listing 3.12 – Interface fournies par le composant `Emergency`

L'implémentation du composant `Emergency` permet de spécifier qu'il possède des interfaces fonctionnelles clientes pour pouvoir le connecter aux autres composants. Le listing 3.13 est un extrait de l'implémentation du composant `Emergency`

```
package applications.emergency.fc.lib;

import org.objectweb.fractal.fraclet.annotations.Component;
import org.objectweb.fractal.fraclet.annotations.Interface;
import applications.emergency.fc.api.*;

@Component(provides = {@Interface(name = "emergency",
  signature = applications.emergency.fc.api.Emergencyclass)})
public class EmergencyImpl implements Emergency{

  @Requires(name = "business")
  private applications.business.fc.api.Business businesss;

  @Requires(name = "socialinsurance")
  private applications.SocialInsurance.fc.api.SocialInsurance socialinsurance;

  EmergencyInformation getInfo(string fullName, long cardID){}
}
```

Listing 3.13 – Extrait de l'implémentation de l'interface du composant `Emergency`

Enfin l'assemblage des composants et la création du composant composite sont fournis par la description FRACTAL ADL du listing 3.14.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL2.0//EN"
          "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

<definition name="applications.emergency.fc.api.EmergencyComposite">
  <interface name="ec" role="server" signature="applications.emergency.fc.api.Emergency"/>

  <component name="Emergency" definition="applications.emergency.fc.lib.EmergencyImpl"/>
  <component name="Business" definition="applications.business.fc.lib.BusinessImpl"/>
```

```
  <component name="SocialInsurance"
             definition="applications.socialinsurance.fc.lib.SocialInsuranceImpl"/>

  <binding client="this.ec" server="Emergency."/>

  <binding client="Emergency.business" server="Business.business"/>
  <binding client="Emergency.socialinsurance" server="SocialInsurance.socialinsurance"/>

</definition>
```

Listing 3.14 – Description de l'assemblage de composants correspondant à la figure 3.4

Assemblage de composants dans SCA. L'assemblage de composants dans SCA se fait par la création d'un composant composite. L'utilisation d'un fichier de description SCDL (*Service Component Definition Language*) [Arc07] permet de définir le composant composite. Ce fichier comprend, de la même manière que dans FRACTAL, les composants impliqués dans la composition ainsi que les liaisons qui sont faites. Les liaisons sont directement définies au niveau des interfaces *references*.

La création d'un composant composite impose la spécification d'interfaces de type *service* pour que celui-ci soit accessible par la suite. Il peut, également, y avoir des interfaces de type *reference* pour permettre de lier ce composant composite à un autre composant. Lors de la spécification de ces interfaces, il est également possible de spécifier le type de liaisons qu'elles supportent. Grâce à la balise *binding*. Il en existe plusieurs dont voici quelques exemple : *binding.ws* spécifie une liaison vers un *Web Service* et utilise donc SOAP, *binding.jms* spécifie une liaison sur Java Message Service et utilise un protocole de fil de messages...

La figure 3.5 illustre la composition dans SCA qui correspond à la même composition réalisée en FRACTAL auparavant.

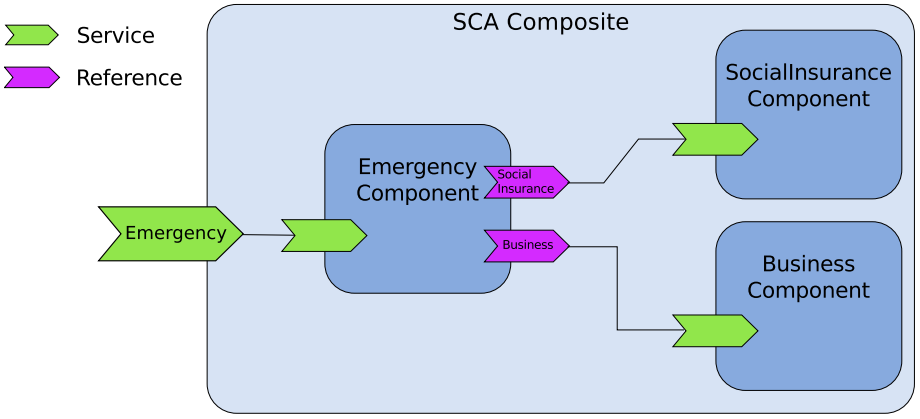


FIGURE 3.5 – Assemblage de composants SCA

Le listing 3.15 décrit le composant composite avec l'ensemble des composants qui se trouve impliqués dans la composition.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  name="Emergency">
  <service name="CalculatorService" promote="EmergencyComponent">
    <interface.java interface="application.sca.emergency.Emergency"/>
    <binding.ws uri="http://localhost:8181/EmergencyService"/>
  </service>

  <component name="EmergencyComponent">
    <implementation.java class="application.sca.emergency.EmergencyImpl"/>
    <reference name="business" target="BusinessComponent" />
    <reference name="socialInsurance" target="SocialInsuranceComponent" />
  </component>
```

```
12
    <component name="SocialInsuranceComponent">
      <implementation.java class="application.sca.socialinsurance.SocialInsuraceImpl"/>
    </component>
      <component name="BusinessComponent">
17    <implementation.java class="application.sca.business.BusinessImpl"/>
    </component>
</composite>
```

Listing 3.15 – Description de l'assemblage de composants correspondant à la figure 3.5

Assemblage de composants dans WCOMP. L'assemblage de composants sous WCOMP se fait par des liaisons entre un port d'événement et un ou plusieurs ports qui correspondent aux méthodes. Les liens spécifient quelle est la source de l'événement et quelle est la méthode appelée. Si cette méthode nécessite l'usage d'un paramètre, il est alors possible de spécifier la méthode du composant émetteur de l'événement qui doit être appelée pour obtenir ce paramètre. La récupération des paramètres se fait par rétro-appel.

L'exécution de l'assemblage est dirigée par l'émission des événements. Dans WCOMP chaque composant ne décrit pas quels événements il écoute, ce sont les liens qui spécifient sur quel événement doit écouter un composant contrairement à JAVABEANS.

Pour permettre de publier un assemblage de composants comme un nouveau service (qui pourrait être importé comme un composant au sein de la plate-forme WCOMP), celui-ci est encapsulé dans un conteneur SLCA (*Service Lightweight Component Architecture*) [HTL+08]. De la même manière que dans les autres plates-formes, il est nécessaire de spécifier quelles sont les méthodes accessibles et quels sont les événements qui sont émis par ce service. Pour ce faire, il est nécessaire d'ajouter des composants sources et des composants puits afin de pouvoir respectivement appeler les méthodes disponibles et des événements.

Le listing 3.16 décrit l'assemblage de composants qui correspond à l'obtention des informations personnelles et professionnelles d'une personne. Pour que le composant `Emergency` obtienne les résultats des autres composants, il est nécessaire que ceux-ci émettent un événement pour le prévenir qu'ils ont fini de récupérer les informations. C'est ainsi que le composant `Emergency` peut, par rétro-appel sur le composant émetteur de l'événement, récupérer la donnée.

```
1  <application>
     <instance type="WComp.Basic.Business_Bean" name="Business" x="32" y="272"/>
     <instance type="WComp.Basic.SocialInsurance_Bean" name="SocialInsurance" x="40" y="168"/>
     <instance type="WComp.Basic.Emergency_Bean" name="Emergency" x="56" y="16"/>

6    <link>
       <source name="Emergency">
         <event name="raisePersonalInfoEvent" />
       </source>
       <destination name="Business">
11       <method name="RetrieveBusinessInfo">
           <parameter type="System.String" call="get_FullName"/>
         </method>
       </destination>
     </link>
16   <link>
       <source name="Emergency">
         <event name="raisePersonalInfoEvent" />
       </source>
       <destination name="SocialInsurance">
21       <method name="RetrieveSocialInfo">
           <parameter type="System.String" call="get_CardID"/>
         </method>
       </destination>
     </link>
26
     <link>
       <source name="Business">
         <event name="raiseInfoEvent" />
       </source>
31     <destination name="Emergency">
```

```
      <method name="set_BusinessInfo">
        <parameter type="Business.BusinessInfo" call="get_BusinessInfo"/>
      </method>
    </destination>
  </link>
  <link>
    <source name="SocialInsurance">
      <event name="raiseInfoEvent" />
    </source>
    <destination name="Emergency">
      <method name="set_SocialInsuranceInfo">
        <parameter type="SocialInsurance.SocialInsuranceInfo" call="get_SocialInsuranceInfo"/>
      </method>
    </destination>
  </link>
</application>
```

Listing 3.16 – Description de l'assemblage de composants au sein de WCOMP

Conclusion

L'ensemble des ces approches s'appuie sur des architectures similaires qui utilisent des composants simples, assemblés au sein de composants composites ou de *containers*. De la même manière que pour les *Web Services*, il est possible de sélectionner un sous-ensemble des données qui vont être présentées. On peut, également, partager une même donnée entre plusieurs sous-composants ou bien encore fusionner le résultat de plusieurs opérations de sous-composants. Il existe une différence majeure avec les *Web Services* et plus particulièrement les orchestrations de *Web Services* : l'ensemble du flot de données et de contrôle est caché au sein de l'implémentation du composant.

3.1.3 Synthèse de la composition fonctionnelle

L'ensemble des approches présenté que ce soient les orchestrations de *Web Services* ou bien les assemblages de composants, s'appuie sur des architectures qui permettent de décrire l'ensemble des opérations proposées par les *Web Services* ou les composants. De plus, ces approches permettent de décrire à la fois le flot de données et le flot de contrôle de manière plus ou moins explicite. En effet, au sein des orchestrations de *Web Services* ou des assemblages faits dans WCOMP, il est possible de connaître précisément le flot de données et de contrôle, ce qui n'est pas le cas au sein des approches SCA et FRACTAL qui cachent ceux-ci au sein de l'implémentation des composants.

Au sein des compositions réalisées, toutes ces approches permettent : (i) de sélectionner un sous-ensemble des données que l'on souhaite fournir, (ii) de partager une donnée en entrée entre plusieurs opérations et (iii) de fusionner le résultat obtenu de plusieurs opérations afin de n'en avoir qu'un seul. Mais, seules les orchestrations de *Web Services* explicitent clairement l'ensemble de ces possibilités. SCA et FRACTAL quant à eux cachent cela au sein de l'implémentation des composants et il n'est possible d'y accéder que si le composant composite le permet (ce qui n'est pas toujours le cas) ; pour WCOMP, il est possible de savoir qu'une donnée est partagée mais la sélection d'un sous-ensemble de données ainsi que la fusion de celles-ci restent cachées au sein de l'implémentation d'un composant. Pour permettre d'expliciter l'ensemble de ces éléments, il est nécessaire d'ajouter des composants intermédiaires qui permettent de faire ressortir la sélection ou la fusion.

3.2 La composition au niveau des IHM

Cette section présente des travaux qui permettent de réaliser la composition d'IHM indépendamment de la partie fonctionnelle de l'application. Ces compositions s'appuient : (i) soit sur la

description de l'IHM (de manière abstraite ou concrète) ou (ii) soit sur la description de l'arbre de tâches. L'ensemble de ces descriptions ont été définies par le projet `Cameleon`, au sein d'un framework de référence : *Cameleon Reference Framework* (CRF) [CCT+03]. Celui-ci décrit quatre niveaux d'abstraction des IHM qui sont utilisés au cours du cycle de développement et qui correspondent à :

- la spécification du domaine et des tâches ;
- la réalisation d'une IHM Abstraite qui décrit l'IHM sans se préoccuper des éléments graphiques ;
- la réalisation d'une IHM Concrète qui spécifie les éléments décrits dans l'IHM Abstraite en associant un objet graphique concret ;
- finalement la réalisation d'une IHM Finale qui correspond à l'IHM qui s'exécute sur une plate-forme donnée.

Les compositions sont donc réalisées sur les trois niveaux les plus abstraits permettant ainsi d'être indépendants d'une technologie ou d'une plate-forme d'exécution donnée.

Les sous-sections qui suivent présentent, dans un premier temps, les compositions réalisées sur la description des IHM (cf. sous-section 3.2.1) et dans un second temps, la composition au niveau des arbres de tâches (cf. sous-section 3.2.2). Enfin, la sous-section 3.2.3 conclut par une synthèse de la composition d'IHM.

3.2.1 La composition de description d'IHM

Ce qui suit présente, dans un premier temps, des langages de description d'IHM qui se situent au niveau abstrait et concret du CRF. Puis, nous présentons les mécanismes de composition mis en œuvre au sein des de ces langages.

Langages de description d'IHM

Chacune des approches utilise son propre langage de description d'IHM. Dans le premier cas, le langage décrit des IHM abstraites. Dans le second, c'est un ensemble de langages qui permet de décrire des IHM abstraites et concrètes. Chacun de ces langages s'appuie sur une représentation arborescente des IHM.

A. Le langage de description d'IHM SunML [PDF03]. Celui-ci permet de décrire une IHM de manière abstraite et est utilisé au sein de l'outil `Amusing`. Il fournit pour ce faire un ensemble de balises permettant cette description. Ce langage s'appuie sur XML et signifie *Simple Unified Natural Markup Language*. Il est basé sur six éléments principaux qui décrivent l'IHM :

- l'*interface* correspond au conteneur principal, c'est cet élément qui va contenir tous les autres ;
- le *dialog* correspond à un sous-conteneur qui peut englober d'autres éléments (il inclut les mêmes éléments que l'interface) ;
- le *menu* décrit des menus. Il peut contenir d'autres menus mais également des *link* qui correspondent aux éléments du menu ;
- le *link* décrit les éléments du menu mais également des éléments d'interaction tel que le bouton lorsque celui-ci est inclus au sein d'un *dialog* ou d'une *interface* ;
- la *list* décrit des listes et contient des *element*s ;
- l'*element* décrit les éléments de bases et permet de représenter des éléments en lecture seule ou lecture-écriture (donc éditable) associé à un certain type (*integer, double, boolean, string* et *image*).

L'ensemble de ces éléments possède un identifiant unique et peut avoir une description ainsi qu'une sémantique. La description est utilisée pour dénommer un élément (ce qui correspond au label associé à un champ de saisie) alors que la sémantique permet de donner un sens à l'élément. Les *menu*s, *link*s et *element*s peuvent posséder une valeur.

Le code présent dans le listing 3.17 illustre la représentation de l'IHM présentée dans la figure 2.2 (à droite). Cette description reprend une partie des éléments présentés auparavant. Les éléments qui doivent recevoir le résultat de l'opération sont mis en mode lecture pour éviter que l'on puisse éditer les valeurs. Seul l'élément *fullName* qui permet la saisie du nom complet est en lecture-écriture. Il y a également un *link* correspondant au bouton pour l'appel de l'opération. Les éléments qui affichent les valeurs de retour de l'opération appelée sont regroupés au sein d'un *dialog* tout comme les éléments associés à l'entrée et à l'appel de l'opération.

```
<?xml version="1.0" encoding="UTF-8"?>
<sunml>
  <interface id="getBusinessInfo" desc="Get Business Info"
             semantic="obtain employee information">
    <dialog id="global">
      <dialog id="ask">
        <element id="fullName" desc="FullName" mode="read-write" type="string"/>
        <link id="getBusinessInfo" semantic="get_business_info"
              value="getBusinessInfo"/>
      </dialog>
      <dialog id="result">
        <element id="fullNameResult" desc="FullName" mode="read" value="-"
                 semantic="fullName" type="string"/>
        <element id="position" desc="Position" mode="read" value="-"
                 semantic="position" type="string"/>
        <element id="email" desc="Email" mode="read" value="-"
                 semantic="email" type="string"/>
        <list id="address" desc="Address" mode="read" semantic="addresses"/>
        <list id="building" desc="Building" mode="read" semantic="buildings"/>
        <list id="office" desc="Office" mode="read" semantic="offices"/>
      </dialog>
    </dialog>
  </interface>
</sunml>
```

Listing 3.17 – Description SUNML de l'IHM pour l'opération `getBusinessInfo`

`Amusing` propose un moteur de rendu qui permet de concrétiser la description réalisée en SUNML afin d'obtenir une IHM Swing. On passe ainsi d'une IHM abstraite à une IHM finale. Les attributs des éléments présents dans la description de l'IHM influencent le rendu final.

Le rendu en `Java` Swing de la description de l'IHM est illustré par la figure 3.6. Les *element*s sont représentés par des champs de texte qui sont accessibles ou non selon qu'ils soient en lecture seule ou en lecture écriture. La description associée à chacun des éléments permet de dénommer les champs de texte et correspondent à des labels. L'ensemble des valeurs qui remplissent les champs de texte est pris dans l'attribut *value* ; le lien (*link*) est rendu en bouton. Enfin les *dialog*s permettent de grouper les éléments entre eux.

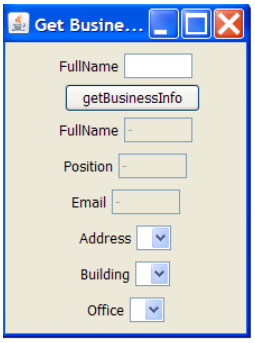


FIGURE 3.6 – Rendu Swing de la description SUNML

B. Les modèles de description d'IHM USIXML [LVM+04]. Ils sont dédiés à la conception des IHM et couvrent les trois premiers niveaux de description d'IHM. Il existe donc un modèle de tâches, un modèle d'IHM Abstraite, un modèle d'IHM Concrète et d'autres modèles pour décrire le contexte, le domaine... Ici, nous nous intéressons plus particulièrement aux modèles de descriptions des IHM que ce soit le modèle d'IHM Abstraite et le modèle d'IHM Concrète.

Le modèle d'IHM Abstraite (AUIMODEL) s'appuie sur deux éléments importants que sont les *Abstract Interaction Object (AIO)* et les *Abstract User Interface Relationship (AUI Relationship)*. Les *AIO* indépendants de toutes modalités et de toutes plates-formes correspondent aux éléments qui composent l'IHM Abstraite. L'IHM Abstraite est décrite comme une hiérarchie d'éléments de la même manière que dans SUNML. On retrouve une notion d'arborescence. Ces *AIO* peuvent posséder différentes facettes qui sont de quatre sortes :

- les facettes d'entrée (*input facet*) décrivent les types d'entrée acceptés par les *AIO*. Les facettes d'entrée possèdent un type de données manipulé, une cardinalité minimale et maximale ;
- les facettes de sortie (*output facet*) décrivent les données qui doivent être présentées à l'utilisateur par un *AIO* ;
- les facettes de contrôle (*control facet*) décrivent les méthodes possibles du noyau fonctionnel qui doivent être déclenchées par un *widget* particulier ;
- les facettes de navigation (*navigation facet*) décrivent les transitions possibles vers un conteneur réalisable grâce à l'utilisation d'un *AIO*.

Il existe également un conteneur qui permet de regrouper des *AIO* entre eux.

Les *AUI Relationship* décrivent des relations abstraites entre des objets d'IHM Abstraite. Une relation peut posséder plusieurs sources et plusieurs cibles. Il existe deux types principaux d'*AUI Relationship* que sont : les relations spatio-temporelles et les transitions de dialogue. Les transitions de dialogue permettent de spécifier une navigation de transition entre un espace d'interaction et un ou plusieurs autres espaces d'interaction avec les possibilités suivantes :

- Suspendu : qui suspend l'espace d'interaction source pour permettre d'activer l'espace d'interaction cible ;
- Résumé : qui permet de résumer un espace d'interaction qui aurait été suspendu au préalable ;
- Désactivé : qui permet de spécifier qu'une source désactive un ensemble de cibles ;
- Activé : qui permet à une source d'activer un ensemble de cibles.

La relation spatio-temporelle caractérise les contraintes physiques entre *AIO* dans leur représentation dans le temps et l'espace. Une seule donnée est nécessaire pour spécifier la relation temporelle alors que les coordonnées en X et en Y sont nécessaires pour spécifier une relation spatiale. Les relations temporelles sont basées sur les relations temporelles d'Allen [All83].

Le code présent dans le listing 3.18 illustre l'utilisation du modèle d'IHM abstraite sur le même exemple que précédemment. On a ici, deux conteneurs qui permettent de grouper les éléments entre eux. Dans le premier conteneur, on retrouve un élément d'entrée pour permettre à l'utilisateur d'effectuer une saisie et un élément de contrôle qui doit être lié à la partie fonctionnelle. Enfin, dans le second conteneur, on a sept éléments qui correspondent aux sorties. On retrouve ainsi les mêmes éléments que dans l'IHM décrite dans SUNML. Les conteneurs utilisés pour regrouper les éléments dans USIXML correspondent aux *Dialog* utilisés dans SUNML.

```
<?xml version="1.0" encoding="UTF-8"?>
<auimodel>
  <abstractContainer id="idaio0" name="Business">
    <abstractContainer id="idaio3" name="ask">
      <abstractIndividualComponent id="idaio5" name="fullName">
        <input id="idaio16" name="idaio16" inputDataType="String" />
      </abstractIndividualComponent>
      <abstractIndividualComponent id="idaio6" name="getBusinessInfo">
        <control id="idaio17" name="idaio17" />
      </abstractIndividualComponent>
    </abstractContainer>
```

```
    <abstractContainer id="idaio4" name="result">
      <abstractIndividualComponent id="idaio7" name="fullName">
        <output id="idaio18" name="idaio18" />
      </abstractIndividualComponent>
      <abstractIndividualComponent id="idaio9" name="position">
        <output id="idaio19" name="idaio19" />
      </abstractIndividualComponent>
      <abstractIndividualComponent id="idaio10" name="email">
        <output id="idaio20" name="idaio20" />
      </abstractIndividualComponent>
      <abstractIndividualComponent id="idaio11" name="address">
        <output id="idaio21" name="idaio21" />
      </abstractIndividualComponent>
      <abstractIndividualComponent id="idaio12" name="building">
        <output id="idaio22" name="idaio22" />
      </abstractIndividualComponent>
      <abstractIndividualComponent id="idaio13" name="office">
        <output id="idaio23" name="idaio23" />
      </abstractIndividualComponent>
    </abstractContainer>
  </abstractContainer>
</auimodel>
```

Listing 3.18 – Description dans le modèle de l'IHM Abstraite d'USIXML de l'IHM pour l'opération `getBusinessInfo`

Le modèle d'IHM Concrète (CUIMODEL) permet de décrire l'apparence et le comportement d'une IHM avec les éléments qui sont perçus par l'utilisateur. L'IHM Concrète est dépendante de la modalité et n'en adresse qu'une à la fois. Elle est, par contre, indépendante de la plate-forme ce qui signifie que les éléments qui constituent l'IHM Concrète correspondent à une abstraction des éléments graphiques que l'on retrouve dans la plupart des boîtes à outils. Les *CIO* (*Concrete Interaction Object*) qui composent l'IHM Concrète sont définis comme des entités perceptibles et/ou manipulables par l'utilisateur (*e.g.* un bouton, une liste, etc.). Les *CIO* sont divisés en deux catégories que sont les conteneurs (*e.g.* fenêtre, table…) et les composants individuels (*e.g.* bouton, menu, champ de texte, etc.).

Le code du listing 3.19 correspond à l'IHM Concrète de l'exemple utilisé précédemment. Ici, on ne retrouve pas d'informations telles que *label* ou *textField* mais on a des *inputText* qui permettent à l'utilisateur de faire ou non de la saisie dans le cas où l'attribut *isEditable* est à faux (équivalent à la lecture seule dans SUNML). Les *label*s correspondent à des *outputText*.

```
<?xml version="1.0" encoding="UTF-8"?>
<uiModel xmlns="http://www.usixml.org"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         id="business_11" name="business"
         creationDate="2010-10-11T09:32:37.481+02:00" schemaVersion="1.6.4">
  <auiModel id="business-aui_11" name="business-aui"/>
  <cuiModel id="business-cui_11" name="business-cui">
    <window id="window_component_0" name="window_component_0"
            width="400" height="350">
      <gridBox id="grid_box_1" name="grid_box_1" cols="3" rows="7">
        <outputText id="fullName" name="FullName" isEnabled="true"/>
        <inputText id="fullName_Value" name="" isEnabled="true" isEditable="true"/>
        <button id="getBusinessInfo" name="getBusinessInfo" isEnabled="true"/>
        <outputText id="fullName" name="FullName" isEnabled="true"/>
        <inputText id="fullNameRes_Value" name="" isEnabled="true" isEditable="false"/>
        <space id="space_7" name="space_7" textColor="#000000" unitGrid="gridUnit"/>
        <outputText id="position" name="Position" isEnabled="true"/>
        <inputText id="position_Value" name="" isEnabled="true" isEditable="false"/>
        <space id="space_10" name="space_10" textColor="#000000" unitGrid="gridUnit"/>
        <outputText id="email" name="Email" isEnabled="true"/>
        <inputText id="email_Value" name="" isEnabled="true" isEditable="false"/>
        <space id="space_13" name="space_13" textColor="#000000" unitGrid="gridUnit"/>
        <outputText id="address" name="Address" isEnabled="true"/>
        <listBox id="address_Value" name="" isEnabled="true" multiple_selection="false"/>
        <space id="space_16" name="space_16" textColor="#000000" unitGrid="gridUnit"/>
        <outputText id="building" name="Building" isEnabled="true"/>
        <listBox id="building_Value" name="" isVisible="true" multiple_selection="false"/>
        <space id="space_19" name="space_19" textColor="#000000" unitGrid="gridUnit"/>
```

```
        <outputText id="office" name="Office" isEnabled="true"/>
        <listBox id="office_Value" name="" isEnabled="true" multiple_selection="false"/>
        <space id="space_22" name="space_22" textColor="#000000" unitGrid="gridUnit"/>
      </gridBox>
    </window>
  </cuiModel>
</uiModel>
```

Listing 3.19 – Description dans le modèle d'IHM Concrète d'USIXMLde l'IHM pour l'opération `getBusinessInfo`

C. Apport de ces descriptions. L'intérêt de ces descriptions est de s'abstraire d'une technologie donnée et de pouvoir cibler différents langages et technologies lors de l'obtention du rendu final. Ceci permet ainsi d'avoir des IHM plastiques [TC99] qui s'adaptent au contexte. Par ailleurs, les IHM abstraites décrites dans USIXML correspondent à des éléments que l'on a au sein du noyau fonctionnel, c'est-à-dire : des entrées, sorties et éléments déclencheurs (qui se rapproche des opérations que l'on retrouve dans le noyau fonctionnel). Les IHM abstraites permettent donc un rapprochement avec le noyau fonctionnel ; placées en amont dans le processus de développement des IHM elles permettent de pouvoir obtenir plusieurs IHM concrètes et par conséquent plusieurs rendus possibles. Le langage SUNML propose quant à lui une description abstraite différente de ce que l'on a dans USIXML qui permet de plus spécifier les éléments manipulés tout en étant indépendant d'une représentation concrète. Il ajoute également une notion de sémantique sur chacun des éléments utiles par la suite lors de la composition.

Composition de descriptions d'IHM

Sur chacun des langages présentés auparavant, il existe des mécanismes de composition. Ils travaillent sur la représentation arborescente des IHM. A SUNML est associé `Amusing` et à USIXML est associé `ComposiXML`. Ces deux outils sont présentés ci-après.

A. La composition dans `Amusing`. Les compositions réalisées au sein d'`Amusing` [PDF03] sont de deux sortes : soit unaires dans le sens où elles n'utilisent qu'une description d'IHM soit binaires car elles utilisent deux descriptions d'IHM.

Il existe trois opérations sur les compositions de type unaire (cf. figure 3.7) :
- l'ajout qui permet d'ajouter un élément dans l'IHM,
- la suppression qui permet de supprimer un élément de l'IHM,
- la duplication qui permet de dupliquer un élément de l'IHM.

Dans le cas de l'ajout, il est nécessaire de spécifier l'endroit où l'on souhaite ajouter l'élément (par défaut, ajouter à la racine). Dans le cas de la duplication, l'identifiant de l'élément est changé afin de conserver l'unicité des identifiants.

Pour les compositions de type binaire (cf. figure 3.8), il existe quatre opérations qui sont : l'union, l'intersection, la soustraction et la substitution. La réalisation de ces opérations nécessite la mise en place d'une fonction d'équivalence. Cette dernière compare : le type des éléments, l'identifiant et la sémantique. Ainsi, deux éléments qui possèdent le même type et le même identifiant sont considérés comme équivalents. C'est également le cas lorsque les identifiants diffèrent, mais que la sémantique est identique.

L'union et l'intersection sont des opérations commutatives qui utilisent la fonction d'équivalence pour : dans le cas de l'union, ne conserver qu'un seul élément en cas de doublon en plus des autres éléments, dans le cas de l'intersection, ne conserver que les éléments équivalents.

La soustraction s'appuie sur l'intersection. Lorsque l'on effectue la soustraction suivante : $IHM_1 - IHM_2$, on supprime de l'IHM_1 l'ensemble des éléments présents dans l'intersection entre l'IHM_1 et l'IHM_2.

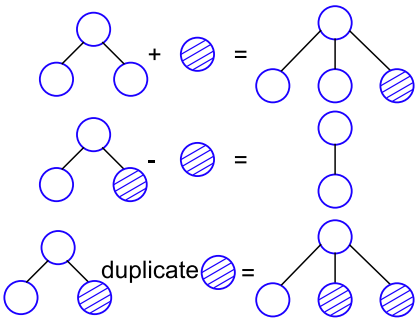


FIGURE 3.7 – Opérateurs de type unaire présents dans `Amusing`

La substitution permet de remplacer une branche ou un élément d'une IHM par une branche ou un élément d'une seconde IHM. Cette opération est comme pour la soustraction non commutative. Pour réaliser la substitution, il faut spécifier sur chacune des IHM l'élément à partir duquel s'effectue la substitution. Ainsi, l'opération de substitution correspond à : $substitution([IHM_1, Elem_1], [IHM_2, Elem_2])$ où $Elem_1$ et $Elem_2$ sont soit un nœud soit une feuille de l'arbre de description d'IHM.

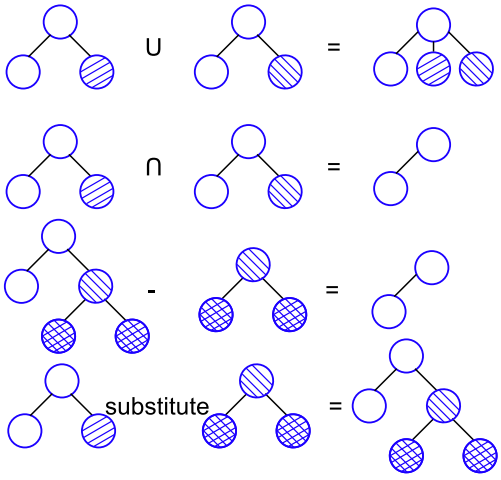


FIGURE 3.8 – Opérateurs de type binaire présents dans `Amusing`

B. La composition dans `ComposiXML`. De la même manière que dans `Amusing`, `ComposiXML` propose deux types d'opérations que sont les opérations unaires et binaires. Ces opérations ont été appliquées dans un premier temps à des descriptions d'IHM Concrète [LV06, LVM06] puis à des descriptions d'IHM Abstraite [LHR+07]. On retrouve dans les opérations de composition proposées une partie des opérations de composition de type binaire présentes dans `Amusing`.

Les opérations de type unaire sont les suivantes :

- la sélection permet de créer une nouvelle description d'IHM à partir d'une description d'IHM et d'un ensemble d'éléments à sélectionner.
- le complémentaire est l'inverse de la sélection dans le sens où l'on retire de l'IHM l'ensemble des éléments sélectionnés. La description de la nouvelle IHM est donc privée de ces

éléments.
- la coupe (*Cut*) permet de supprimer un sous-arbre de l'arbre de description en spécifiant le nœud à partir duquel on souhaite effectuer cette suppression. Cette opération prend donc en paramètre une description d'IHM et un nœud .
- la projection correspond à l'extraction d'un sous-arbre de la description de l'IHM à partir d'un nœud spécifique. Le résultat est donc une nouvelle IHM qui ne contient que ce sous-arbre.

Pour les opérations binaires, on retrouve, comme dans SUNML, les opérations d'union (*Normal Union*), d'intersection et de différence. La différence est représentée par deux opérations qui sont la différence gauche et la différence droite (qui permet ainsi de changer l'IHM sur laquelle on soustrait les éléments). En plus de ces opérations, il existe également : la fusion, l'union unique et la jointure.

La fusion permet d'ajouter à l'union les éléments présents dans l'intersection, et ce afin de conserver tous les doublons. L'union unique (*Unique Union*), au contraire, retire l'ensemble des éléments présents dans l'intersection pour ne conserver que les éléments qui sont présents uniquement dans une IHM. La jointure (*Join*) est utilisée pour concaténer l'ensemble des nœuds des deux IHM en fonction des nœuds communs.

En outre ces opérations de compositions, il existe des opérations qui permettent de comparer deux IHM. Ces opérations sont : la similarité, l'équivalence et la subsomption. La similarité teste la structure des IHM sans se préoccuper des données alors que l'équivalence compare à la fois la structure et les données. La subsomption, quant à elle, permet de vérifier qu'une des IHM est incluse dans l'autre.

La figure 3.9 est extraite de l'article [LV06] et reprend de manière imagée l'ensemble des opérations présentes au sein de `ComposiXML`.

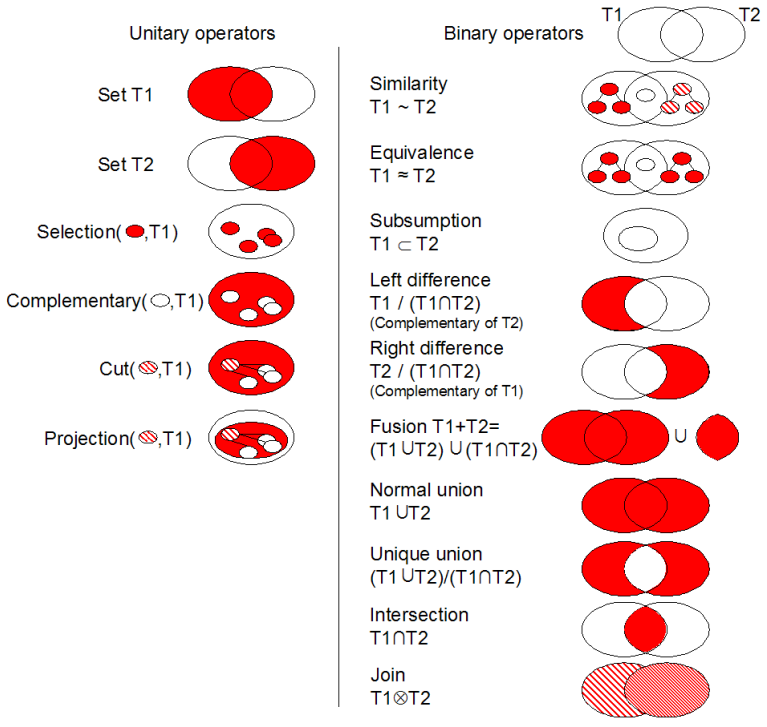


FIGURE 3.9 – Ensemble des opérations de compositions présentes dans `ComposiXML`

Conclusion

Les langages de descriptions d'IHM sur lesquels s'appuient les outils permettant la composition, décrivent les IHM de manière plus ou moins abstraite. USIXML propose deux langages décrivant des IHM abstraites (en terme d'entrée, sortie, de contrôle et de navigation) et des IHM concrètes dans laquelle on retrouve une description des *widgets* utilisés. SUNML décrit, quant à lui, des IHM abstraites, mais de manière un peu plus précise qu'USIXML puisque l'on introduit la notion de liste, d'élément, de liens...Cependant de manière générale, il est possible de retrouver dans chacune de ces descriptions abstraites la notion d'entrée, sortie et d'événement qui doit faire le lien avec la partie fonctionnelle. De plus, ces langages décrivent uniquement la partie Vue retrouvée dans les architectures de type MVC et permettent donc de bien séparer les éléments de l'application.

Les opérations de composition sont similaires entre `Amusing` et `ComposiXML`. Toutes ces opérations réalisent des compositions structurelles d'IHM. Pour que ces compositions soient possibles, il est nécessaire que les descriptions d'IHM fournissent des éléments de comparaison sur lesquels s'appuyer. Les compositions réalisées ne spécifient pas l'utilisation qui est faite des données. Ainsi, lorsque deux éléments d'IHM sont fusionnés et qu'il n'en reste qu'un seul, il n'est pas possible de savoir si : (i) les données à afficher sont fusionnées ou (ii) une seule donnée est affichée.

Ces compositions permettent ainsi de : (i) sélectionner les éléments graphiques que l'on souhaite conserver et (ii) de "fusionner" (dans le sens ne conserver qu'un élément graphique sur les deux présents). Néanmoins l'ensemble de ces opérations de composition ne concerne que les éléments graphiques et ne se préoccupe pas des informations qui sont affichées au sein de ces éléments ou bien encore les informations qui doivent être saisies. Il n'est donc pas possible de connaître la manière dont sont gérées les données à afficher ou à saisir.

3.2.2 La composition d'arbres de tâches

La composition d'IHM peut se faire en amont dans le processus de développement au niveau du modèle de tâches. Ce type de composition est à cheval entre la composition d'IHM et la composition d'applications. En effet, le modèle de tâches intègre les tâches systèmes mais il n'établit pas de liaisons avec la fonctionnalité concrète et n'a donc aucune d'implication sur la partie fonctionnelle.

Deux approches différentes utilisent le modèle de tâches pour réaliser la composition, soit en les fusionnant [LLB07], soit en les enrichissant pour permettre la composition de celles-ci [BB08].

Avant de présenter ces approches, ce qui suit définit le modèle de tâches.

Définition

Le modèle de tâches est apparu pour analyser le travail effectué par des employés dans l'entreprise, en décrivant les tâches qu'ils devaient réaliser. Maintenant, il fait partie intégrante du processus de création d'une application, principalement pour la réalisation de l'IHM. Le modèle de tâches permet de décrire les activités logiques que doit réaliser l'utilisateur pour atteindre son objectif, ainsi que l'enchaînement des tâches les unes par rapports aux autres ; grâce à des opérateurs de choix, de séquentialité ou de parallélisme. De nombreux modèles ont été définis pour décrire les tâches tels que : `HTA` (*Hierarchical Task Analysis*) [AD67, Ann03], `CTA` (*Cognitive Task Analysis*) [HSH90], `GOMS` (*Goals, Operators, Methods, and Selection rules*) [JK94] ou encore `CTT` (*ConcurTaskTree*) [PMM97].

`CTT` cible, plus particulièrement, la description de tâches pour des systèmes interactifs. Il introduit donc des tâches plus spécifiques à la conception des IHM. Ainsi on retrouve :

– des tâches utilisateur (*user tasks* qui ne demandent pas d'interactions avec le système mais qui, par contre, requièrent une activité physique ou cognitive ;

– des tâches applicatives (*application tasks*) qui sont exécutées complètement par le système ;
– des tâches d'interaction (*interaction tasks*) qui sont réalisées par l'utilisateur en interaction avec le système ;
– des tâches abstraites (*abstract tasks*) qui sont des tâches complexes à réaliser ne pouvant être décrites par une des trois tâches précédentes. Une tâche abstraite est réalisée contient un ensemble de tâches (concrètes ou abstraites).

[Pat00] introduit un ensemble d'opérateurs temporels pour spécifier l'enchaînement des tâches. Tout d'abord, la hiérarchie de tâches permet de décomposer une tâche complexe (donc une tâche abstraite) en un certain nombre de sous-tâches. Ces sous-tâches sont liées par des opérateurs temporels qui permettent d'exprimer : l'activation d'une tâche par une autre, le choix, l'activation avec passage de données, la concurrence, l'indépendance, la désactivation et suspendre/reprendre.

Pour permettre la création de ces arbres de tâches, l'outil `CTTe` [MPS02] a été dévéloppé. Il permet d'obtenir une représentation graphique du modèle de tâches. La figure 3.10 représente le modèle de tâches associé à l'application `Business` permettant d'obtenir les informations professionnelles à partir du nom de la personne.

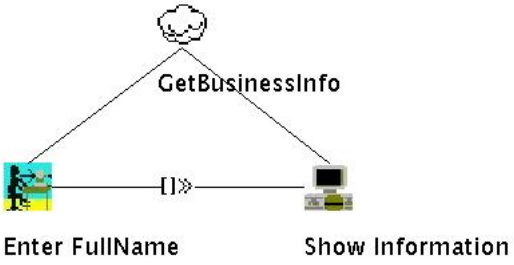


FIGURE 3.10 – Exemple d'arbre de tâches décrit avec `CTTe` pour l'application permettant d'obtenir les informations professionnelles à partir du nom de la personne

Le modèle de tâches est comparable, en quelque sorte, au *workflow* présent dans les compositions fonctionnelles. Le *workflow* est spécifique aux tâches systèmes et décrit précisément les échanges et les enchaînements entre les tâches contrairement au modèle de tâches qui est plus haut niveau et qui englobe également les tâches utilisateurs.

Le modèle de tâches est un élément clé du processus de création d'une IHM. Ce modèle permet de spécifier les différentes actions de l'utilisateur ainsi que l'ordre dans lequel elles doivent être réalisées. Il permet d'obtenir facilement une première IHM Abstraite (raffinement du modèle de tâches en modèle d'IHM Abstraite). Ce modèle a été utilisé dans deux approches pour permettre la composition.

Composition au niveau du modèle de tâches

Les deux approches présentées ci-après, s'appuient sur l'utilisation de modèles de tâches. Dans la première approche, ce modèle reste assez simple et décrit l'exécution des tâches et quelles sont les tâches actives. On ne distingue pas différents types de tâches. Cette approche compose de manière dynamique les IHM. La seconde approche s'appuie sur le modèle `CTT` enrichi et cherche à fusionner deux arbres de tâches. Cette approche se place en amont dans le cycle de conception des IHM avant même d'avoir l'IHM abstraite. Des outils permettent alors de raffiner le modèle de tâches pour obtenir une IHM abstraite.

Composition d'IHM dirigée par le modèle de tâches. L'approche proposée par Betermieux et Bomsdorf dans [BB08] utilise les arbres de tâches pour diriger la composition des IHM. La

composition d'IHM qu'ils réalisent ne correspond en rien aux travaux présentés auparavant dans le sens où il n'y a pas d'opérations de composition. Ici, la composition correspond à une création d'IHM à partir d'éléments que l'on assemble.

Cette approche s'appuie sur un modèle de tâches dédié à la représentation des tâches pour les applications web qui s'appuie sur `CTT` : `WTM` (*WebTaskModel*) [Bom07]. Une tâche peut posséder une cardinalité qui stipulant si celle-ci est optionnelle ou non. Un lien dirigé entre deux tâches permet de spécifier l'ordre d'exécution. Chaque tâche possède une machine à état spécifique composée de six états : initié (*Initiated*), suspendu (*Suspended*), exécution (*Running*), achevé (*Completed*), terminé (*Terminated*) et passé (*Skipped*). Pour qu'une tâche soit initiée, il est nécessaire que toutes les préconditions (qui correspondent aux conditions et relations temporelles) soient remplies ; elle peut alors passer à l'état exécuté.

La composition est rendue possible grâce à l'enrichissement du modèle de tâches par des fragments d'IHM associés à chacune des tâches, mais également à un état qui est soit en exécution soit initié (dans le cas d'une tâche optionnelle, l'utilisateur peut décider de passer cette tâche et donc son état passera d'initié à passé). Ces fragments d'IHM sont utilisés pour composer l'IHM à l'exécution et fournir une IHM qui correspond à l'état courant de l'arbre de tâches.

Pour permettre de réaliser la composition, le modèle de tâches est rendu exécutable dans le sens où, il est possible de connaître l'état d'une tâche à tout instant. Ce modèle de tâches exécutable est utilisé par un contrôleur de tâches qui vérifie que le modèle est cohérent par rapport aux états qu'ils possèdent. Ainsi il s'assure que dans une séquence, une seule tâche de cette séquence est active à la fois.

L'arbre de tâches est alors restreint aux seules tâches actives dont les fragments d'IHM sont utilisés par le compositeur de page (*page composer*). Ce dernier compose l'ensemble des fragments d'IHM afin d'obtenir une IHM Abstraite transformée en une IHM Finale au travers d'un moteur de rendu. Ils obtiennent ainsi une page HTML.

Ainsi, l'évolution du statut des tâches au cours de l'exécution dirige la composition des fragments d'IHM pour la création de la page web associée aux tâches courantes.

Dans le cas de la composition des scénarios, l'arbre de tâches comporte deux tâches qui sont la saisie du nom et la saisie du numéro de carte. La figure 3.11 illustre le modèle de tâches simplifié utilisé dans cette approche.

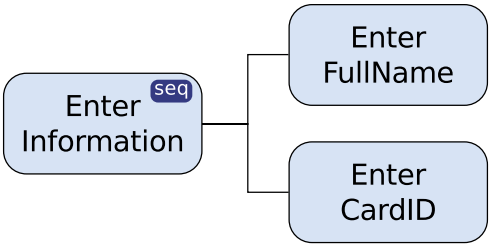


FIGURE 3.11 – Modèle de tâches simplifié qui décrit la saisie du nom et du nuémro de carte

La figure 3.12 illustre quant à elle les annotations mises en place sur les tâches qui sont utilisées à l'exécution pour composer l'IHM.

La composition proposée ici n'est pas une composition mais plutôt une construction d'IHM. Cette construction a pour avantage d'intervenir à l'exécution. La composition vient de l'assemblage de plusieurs fragments d'IHM pour former l'IHM sur laquelle va interagir l'utilisateur. Le modèle de tâches utilisé dans cette approche est une version très simplifiée qui ne distingue qu'un seul type et non plusieurs, comme dans CTT. On perd donc la notion de tâches systèmes.

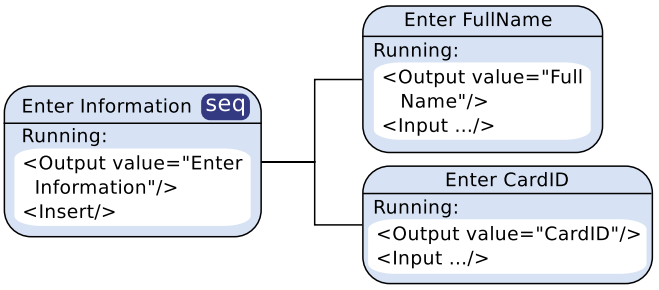


FIGURE 3.12 – Annotation du modèle de tâches avec les fragments d'IHM

Fusion de modèles de tâches. L'approche proposée par Lewandowski *et al.* [LLB07] permet de fusionner des modèles de tâches. Pour ce faire, les auteurs utilisent une approche orientée tâche où les tâches sont liées aux composants ce qui aboutit à la préservation du lien avec le code source [BLT07]. Cette approche orientée tâches permet de conserver le modèle de tâches durant tout le cycle de conception.

La réalisation de la composition des modèles de tâches utilise les concepts présents dans `ComposiXML` qui permettent de réaliser la composition d'arbres XML. Il est nécessaire de convertir, dans un premier temps, le modèle de tâches en arbre XML. L'union de deux arbres de tâches correspond à l'opération de composition réalisée. Pour ce faire, les auteurs recherchent des points de similitude dans les arbres de tâches pour permettre la fusion des tâches.

La similitude ne signifie pas que les sous-tâches doivent contenir exactement les mêmes informations. En effet, il peut y en avoir plus par exemple. Les tâches identiques sont fusionnées pour n'en conserver qu'une seule, les autre tâches qui diffèrent sont conservées et la logique temporelle est ajoutée afin de décrire l'enchaînement des tâches. Toutes ces sous-tâches sont regroupées au sein d'une même tâche abstraite.

Certaines tâches sont regroupées cependant d'autres tâches demeurent distinctes. Malgré cela, il est nécessaire d'ajouter des notions temporelles de choix, par exemple, pour permettre de proposer les tâches devant être utilisées par la suite.

L'arbre de tâches qui résulte de la composition est alors raffiné pour obtenir une IHM. Cette composition n'est pas directement liée aux IHM puisqu'elle est réalisée en amont dans le cycle de conception des IHM.

Malgré l'attachement du modèle de tâches à des composants afin de pouvoir le conserver durant toutes les phases de développement, les tâches systèmes ne sont pas exploitées. Par ailleurs, la fonction de similitude qui permet de rechercher des tâches abstraites à composer n'est pas clairement explicitée.

Conclusion

Le modèle de tâches se place en amont dans le cycle de développement des IHM et permet de décrire les tâches d'interaction de l'utilisateur avec le système ainsi que les tâches systèmes. Par raffinement, il est possible de générer une IHM à partir des tâches d'interaction ; ceci implique la mise en place de règles de transformations. Ceci peut s'appuyer sur la méthodologie mise en place au sein du CRF. Malgré la présence des tâches systèmes, les approches qui utilisent les arbres de tâches ne les prenant pas en considération, il n'y a aucun impact sur la partie fonctionnelle.

Les deux approches diffèrent dans le sens où, l'approche de [BB08] réalise la composition à l'exécution, en annotant l'arbre de tâches par des fragments d'IHM, permettant une composition/construction d'IHM en fonction des tâches qui s'exécutent tandis que celle de [LLB07] se

place au niveau de la conception, pour réaliser la composition des arbres de tâches en réalisant une union de ceux-ci.

Les possibilités offertes avec les arbres de tâches permettent de décrire des compositions telles que celle décrite en section 2.2.2. On se rapproche également des possibilités offertes par la composition fonctionnelle comme celles trouvées dans les orchestrations de *Web Services*.

3.2.3 Synthèse de la composition d'IHM

La composition d'IHM peut se faire à différents niveaux d'abstraction du CRF. Il est possible de les réaliser au niveau du modèle de tâches ou bien au niveau des descriptions d'IHM qu'elles soient Abstraites ou Concrètes. Les compositions réalisées au niveau des descriptions d'IHM s'appuient sur l'usage d'opérateurs de composition tels que l'union, l'intersection... [PDF03, LV06, LVM06, LHR+07]. Ce sont des compositions structurelles qui ne prennent pas en considération la partie fonctionnelle. Elles ne permettent donc pas de pouvoir spécifier l'utilisation qui est faite des données et d'ainsi exprimer la fusion de plusieurs données ou bien encore la sélection d'une donnée en particulier. Ce qui nécessite donc un travail supplémentaire à réaliser de la part du développeur. Les compositions qui utilisent le modèle de tâches permettent de réaliser une sélection et une union des tâches présentes, et de ce fait, des éléments d'IHM que l'on peut obtenir par raffinement [LLB07], ou dans l'autre cas, de construire une IHM à l'exécution en fonction des tâches actives [BB08]. Les modèles de tâches intègrent la description de tâches systèmes qui ne sont pas prises en considération lors de la composition. De la même manière que pour la composition des descriptions d'IHM, il n'est pas possible de pouvoir exprimer une composition des données. Seule la suppression de tâches redondantes (dans le cas de [LLB07]) est possible.

Ces approches distinguent, dans le cas des compositions de description d'IHM, la partie Vue et, dans le cas des compositions au niveau des tâches, l'ensemble des éléments du MVC. Ensuite, chacune des approches propose de réaliser des compositions permettant de sélectionner ou de fusionner des éléments. Par contre, la fusion correspond plutôt à la conservation d'un seul élément parmi plusieurs sans se soucier du devenir des données. Seul le cas de l'utilisation des arbres de tâches permet d'avoir une fusion (*i.e.* une union) qui permet de partager la donnée au niveau du fonctionnel, par la suite.

3.3 La composition d'applications

La composition au niveau fonctionnel et au niveau IHM ainsi que les éléments nécessaires à la réalisation de ces compositions ont été décrits dans les sections précédentes. Les sous-sections suivantes décrivent la composition d'applications où la partie fonctionnelle et la partie IHM sont prises en compte dans la réalisation de cette composition. La sous-section 3.3.1 présente des travaux sur la composition d'applications dirigée par les IHM. La sous-section 3.3.2 présente des travaux sur la composition d'applications dirigée par la partie fonctionnelle. Enfin la sous-section, 3.3.3 conclut cette section sur la composition d'applications.

3.3.1 Compositions dirigées par l'IHM

Ce qui suit présente des approches réalisant la composition d'applications dirigée par les IHM. Ces compositions s'appuient sur différentes approches qui sont : (i) l'utilisation des *Mashups*, (ii) l'utilisation de la planification et (iii) l'utilisation de services annotés. Ces approches permettent de réaliser des compositions plus ou moins complexes (juxtaposition d'IHM, utilisation de données d'un élément d'IHM dans un autre, fusion...).

Utilisation des *Mashups*

Les approches présentées ci-après s'appuient sur l'utilisation de *Mashups* [Mer06]. Les *Mashups* sont des applications web légères qui intègrent de multiples données et fonctions au sein d'une application. Les *Mashups* sont généralement des applications orientées navigateur web, pour que les utilisateurs puissent facilement accéder à ceux-ci. Leur architecture s'appuie sur trois éléments qui sont : un fournisseur de contenus, un site d'hébergement du *Mashup* et un navigateur web comme consommateur.

L'intérêt principal des *Mashups* est de fournir une vue haut niveau des applications au travers de leurs IHM, afin de permettre à des utilisateurs finaux de réaliser la composition. Les paragraphes suivants présentent différentes approches associées aux *Mashups*.

Composition au sein d'un *Dashboard*. Cette composition permet à l'utilisateur de créer son propre environnement afin de réunir différentes sources d'informations (application). C'est ce que l'on retrouve dans *iGoogle* [1] ou *Netvibes* [2]. Dans ces approches, il n'est possible de réaliser que des **juxtapositions** d'applications qui restent indépendantes les unes des autres. La figure 3.13 illustre l'utilisation de deux applications (l'une de météo et l'autre d'agenda) au sein d'*iGoogle*.

FIGURE 3.13 – Deux applications juxtaposées au sein d'*iGoogle*

L'intérêt est l'utilisation même des *Mashups* qui s'appuient sur un découpage de type MVC ainsi que la possibilité d'agréger plusieurs applications sur une même fenêtre, première étape de composition d'applications. L'incapacité de fusion d'éléments ainsi que d'échanges de données entre applications (restant donc indépendantes) sont les inconvénients majeurs de cette approche. En effet, il est seulement possible de juxtaposer les IHM des applications.

Composition au sein de *Yahoo !Pipe*. *Yahoo ! Pipe* [3] donne la possibilité à l'utilisateur de connecter de manière manuelle plusieurs sources d'informations. Ici, l'utilisation du *end-user programming* permet à des utilisateurs "non experts" de pouvoir faire l'union de plusieurs sources d'informations, de créer des boucles, d'extraire une sous-partie des données... Pour ce faire, un grand nombre d'opérateurs pour la manipulation des données est fourni Cette composition s'appuie sur la réalisation d'un *pipe*. Ce *pipe* permet décrire l'enchaînement des opérations que l'on souhaite effectuer. La figure 3.14 illustre la création d'un *pipe* faisant l'union des données obtenues par ces deux *Web Services* utilisés comme sources d'informations.

1. http://www.google.com/ig
2. http://www.netvibes.com/
3. http://pipes.yahoo.com/

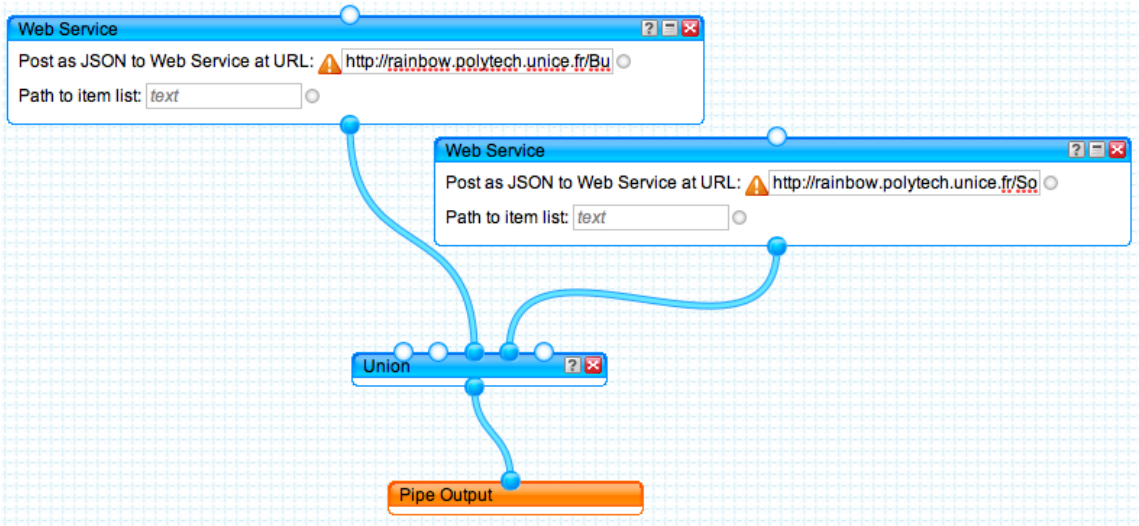


FIGURE 3.14 – Création d'un *pipe* qui utilise deux *Web Services* pour obtenir les informations d'une personne

L'intérêt de cette approche est de permettre à l'utilisateur de contrôler l'utilisation des données qui proviennent de flux (comme les flux RSS). Il est possible de fusionner plusieurs sources de données afin de n'en obtenir qu'une seule ou bien de restreindre les données à un sous-ensemble (sélection). Le problème soulevé par l'approche *Yahoo !Pipe* est l'incapacité de maîtriser le rendu au niveau de l'IHM ainsi que l'incapacité à pouvoir introduire de nouveaux opérateurs. On est donc limité à ceux fournis par *Yahoo !Pipe*.

IHM web orientée services. Les travaux de Pietschmann *et al.*[PVM09, Pie09] utilisent les *Mashups* pour permettre la composition dynamique d'interfaces utilisateurs web orientées services. Cette composition s'appuie sur l'utilisation de services d'IHM. Visualiser les IHM comme des services, permet de créer des applications composites définies comme un assemblage d'un service composite et d'un service d'IHM. Les composants d'IHM présents peuvent être liés à l'exécution, de la même manière que les services. La création de l'IHM passe par un service d'intégration qui a pour objectif de trouver, d'une part, les meilleurs services d'IHM (correspondant aux requis de l'application et au contexte) et, d'autre part, de créer les liaisons spécifiques à la plate-forme d'exécution. Ces informations sont fournies à la partie exécution de la plate-forme. Durant l'exécution, le service de contexte continue de monitorer l'ensemble du système ainsi que les interactions de l'utilisation afin de permettre une adaptation dynamique de l'application à l'exécution.

Utilisation de la planification

L'approche de Gabillon *et al.* [GCF08a] est de réaliser la composition de systèmes interactifs en utilisant la planification. Un système interactif [SGC+07] correspond à un graphe de modèles incluant le noyau fonctionnel, l'IHM (avec les trois premiers niveaux d'abstraction de CRF) ainsi que le contexte d'usage(exprimé par le triplet <Utilisateur, Plate-forme, Environnement>) . Des *mappings* sont également définis pour permettre de lier le noyau fonctionnel à l'IHM ainsi que le noyau fonctionnel et l'IHM au contexte d'usage et plus particulièrement à la plate-forme. L'objectif est de réaliser la composition en réponse à un besoin utilisateur exprimé en langage naturel.

La réponse du système est un ensemble de plans qui répondent aux besoins formulés par l'utilisateur.

Pour obtenir les plans possibles à partir des besoins utilisateurs qui ont été exprimés en langage naturel, il est nécessaire de réaliser une première transformation aboutissant à l'obtention d'un modèle de tâches. Ce dernier est ensuite transformé en un problème de planification à partir duquel, le planificateur est capable de calculer un ensemble de plans qui répondent aux tâches exprimées par l'utilisateur. Ces plans sont alors transformés en modèles de tâches qui sont eux-mêmes transformés en IHM. L'ensemble des ces transformations s'appuient sur l'utilisation de l'ingénierie dirigée par les modèles [GCF08b].

Le calcul des plans s'appuie sur l'utilisation de l'ensemble des systèmes interactifs à disposition qui permettent de réaliser la tâche demandée par l'utilisateur. Le calcul des plans prend également en considération le contexte d'usage dans lequel se trouve l'utilisateur afin d'adapter, par exemple, le résultat obtenu au dispositif utilisé.

Malgré l'utilisation de services interactifs qui comportent à la fois la partie IHM et la partie fonctionnelle, **seule la partie IHM est composée sans implication sur la partie fonctionnelle**.

Utilisation de services annotés

ServFace est un projet européen [4] qui propose d'utiliser des *Web Services* annotés par des éléments d'IHM pour réaliser la composition. Ces annotations sont réalisées par un expert en IHM. Pour réaliser les annotations, une première phase consiste en inférence à partir de la description du *Web Service* et des éléments qui sont présents au sein du WSDL [IJH+09]. L'inférence qui est faite permet d'obtenir une première version des éléments d'IHM qui correspondent aux paramètres en entrée et en sortie pour chacune des opérations présentes. L'inférence réalisée s'appuie sur l'utilisation d'une équivalence entre type de données présent dans les opérations et un élément graphique. Par exemple, à un booléen correspond une *checkbox*. A partir de ces éléments, l'expert peut retravailler sur les éléments inférés, afin de les enrichir, pour obtenir une meilleure IHM que celle obtenue par inférence. L'ensemble des annotations ajoutées par l'expert en IHM est alors conservé en complément du WSDL. Dans le cas du *Web Service* qui permet d'obtenir les informations professionnelles d'une personne (*BusinessService*), on a deux opérations avec un ensemble de paramètres. Le fichier d'annotations correspondant à ce service est décrit par le listing 3.20.

```
<annotation>
  <operation name="getBusinessInfo">
    <parameter type="in">
      <labelContent name="FullName"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>TextField</annotationProperty>
    </parameter>
    <parameter type="out">
      <labelContent name="fullName"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>Label</annotationProperty>
    </parameter>
    <parameter type="out">
      <labelContent name="email"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>Label</annotationProperty>
    </parameter>
    <parameter type="out">
      <labelContent name="position"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>Label</annotationProperty>
    </parameter>
    <parameter type="out">
      <labelContent name="address"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>List</annotationProperty>
```

4. http://www.servface.eu/

```
    </parameter>
    <parameter type="out">
      <labelContent name="building"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>List</annotationProperty>
    </parameter>
    <parameter type="out">
      <labelContent name="office"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>List</annotationProperty>
    </parameter>
  </operation>
  <operation name="getBusinessInfo">
    <parameter type="in">
      <labelContent name="FullName"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>TextField</annotationProperty>
    </parameter>
    <parameter type="out">
      <labelContent name="addresses"/>
      <uicomponent type="TextAnnotation"/>
      <annotationProperty>List</annotationProperty>
    </parameter>
  </operation>
</annotation>
```

Listing 3.20 – Fichier comportant les annotations du *Web Service* de l'application *Business* par des éléments d'IHM

A partir des interfaces des *Web Services* et des annotations associées à ceux-ci, deux approches sont proposées. Elles s'appuient respectivement sur l'*end-user programming* en utilisant un outil qui permette la manipulation des IHM pour la réalisation de la composition et sur l'utilisation de modèles de tâches extraits des annotions des *Web Services*.

Composition au travers des éléments de l'IHM. Nestler *et al.* [NFPS09] présentent une approche de composition de *Web Services* annotés au niveau présentation, qui permet à des utilisateurs "non experts" de manipuler directement les IHM pour réaliser la composition. Pour ce faire, l'utilisateur dispose de l'ensemble des opérations présentes au sein des *Web Services* qui ont été annotés. L'usage d'une de ces opérations fait alors apparaître l'IHM correspondante. L'utilisateur peut alors créer graphiquement des liaisons entre deux composants graphiques afin d'exprimer un transfert de données. La méthodologie mise en œuvre [FJN+09] pour permettre la réalisation de cette composition s'appuie sur trois étapes : (i) l'annotation des *Web Services* (qui a été présentée auparavant), (ii) la composition de ces *Web Services* annotés et (iii) la génération de l'IHM finale à l'exécution. L'étape de composition de services construit une application composite par utilisation de services annotés. Durant la composition, l'utilisateur peut enrichir les éléments en ajoutant des notions de *layout* ou d'enchaînements de pages (pour le côté IHM) mais peut également définir le flot de données. Une transformation de type *model-to-code* est appliquée sur la composition réalisée par l'utilisateur afin de transformer les annotations en une IHM finale qui interagit avec le (ou les) *Web Service*(s) impliqué(s) dans la composition.

Cette approche, proche de ce qu'il est possible de réaliser en utilisant les *Mashups*, comme dans *iGoogle*, permet d'aller plus loin en permettant d'**exprimer des échanges de données** entre composants graphiques tout en donnant la possibilité à un utilisateur non expérimenté la possibilité de créer sa propre application par composition. Par contre, il n'y a **pas d'impact sur la partie fonctionnelle** dont l'ensemble des *Web Services* reste indépendant. Ainsi, les *Web Services* sont utilisés uniquement pour permettre d'associer une IHM à une fonctionnalité. L'ensemble de la composition reste au niveau des IHM.

Composition au travers du modèle de tâches. Feldmann *et al.* [FHSS09] propose d'utiliser les *Web Services* annotés pour réaliser la composition dirigée par le modèle de tâches. Les tâches systèmes présentes au sein du modèle sont représentées par une opération d'un *Web Service* annoté.

Ces tâches sont appelées tâches annotées. Elles sont créées lors de l'utilisation d'une opération au sein du modèle de tâches. L'ensemble des tâches interactives est déduit des annotations associées à l'opération. Les associations entre les tâches systèmes et interactives sont déduites à partir des annotations que l'opération possède. De ces informations, il est ainsi possible de déduire le flot de données qui correspond aux échanges de données entre les tâches ainsi que le flot de contrôle qui spécifie l'ordre et les conditions pour exécuter une tâche. Les tâches interactives ainsi que les associations sont ensuite réifiées afin d'obtenir l'IHM qui permet l'usage de cette composition.

La figure 3.15 représente le modèle de tâches dérivé des tâches annotées associées aux *Web Services BusinessService* et *SocialInsuranceService*. Deux opérations sont utilisées dans cet arbre de tâches. L'une ou l'autre des opérations peut être exécutée, c'est donc un choix qui est laissé à l'utilisateur. Les tâches interactives : *Enter FullName*, *Enter CardID* et les deux *Show Result* sont ajoutées grâce aux annotations présentes sur chacune des opérations. L'approche utilise également la

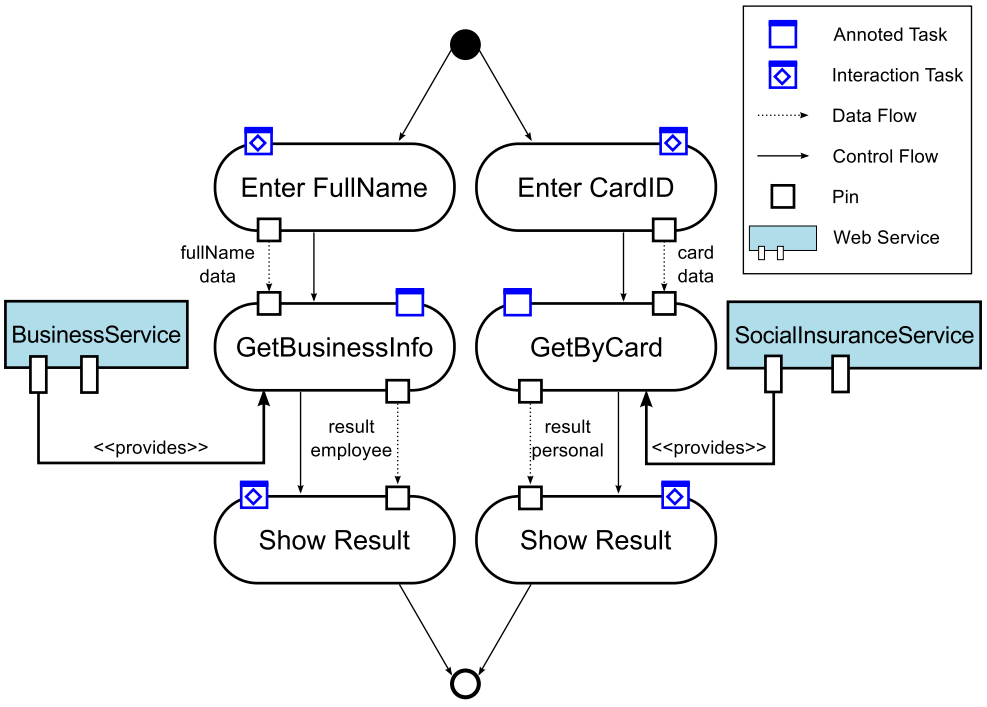


FIGURE 3.15 – Modèle de tâches dérivé des tâches annotées correspondant à l'utilisation de l'une ou l'autre des opérations

composition d'IHM définie dans `ComposiXML` et, plus particulièrement, l'opération d'union afin de pouvoir regrouper des tâches interactives et de supprimer les doublons de ces IHM (cf. section 3.2.2 [LLB07]).

Cette approche utilise les arbres de tâches pour réaliser la composition ; les opérateurs d'enchaînement de tâches y sont ainsi retrouvés. L'intérêt ici est d'obtenir l'ensemble des tâches interactives à partir des tâches systèmes annotées. Cette approche permet de décrire des compositions proches des *workflow* que l'on peut avoir lors de la description d'orchestration de *Web Services*. Par contre, on ne retrouve pas **la richesse qui est fournie par WS-BPEL, par exemple, où l'on peut spécifier complètement le flot de données**. Ici, seules les tâches sont manipulées. L'**union des tâches interactives permet de supprimer les redondances sans possibilité de fusionner** les éléments (et donc les données) au sein de celles-ci.

3.3.2 Compositions dirigées par le fonctionnel

La sous-section précédente a montré la composition dirigée par la composition d'IHM. Maintenant, nous allons voir des approches qui permettent de réaliser la composition d'applications dirigée, cette fois-ci, par la composition de la partie fonctionnelle. La première approche présentée se nomme SOAUI et intègre la notion d'IHM au sein d'une architecture orientée services (SOA : *Service Oriented Architecture*). La seconde approche considère les fonctionnalités volatiles comme point de départ de la composition dans les applications web.

Intégration des IHM dans SOA

L'approche de [THEC08] propose d'intégrer les IHM au sein de SOA. De la même manière qu'il est possible de publier des services au sein d'un annuaire de services ou de (re-)composer des services dynamiquement, les auteurs proposent de voir une IHM comme un service. Celles-ci doivent être publiées au sein d'un annuaire et composées dynamiquement. Pour ce faire, il est nécessaire que les IHM suivent une conception particulière. En effet, celles-ci s'appuient sur des descriptions comme UIML ou XAML. Mais elles possèdent également : (i) une ontologie, (ii) un profil des composants visuels, (iii) un profil de données et (iv) une description du *workflow* de l'IHM (décrit dans ce qui suit). L'IHM possède, au moins, une description de la catégorie d'IHM à laquelle elle appartient. Ces catégories au nombre de cinq sont :

- collection de données : pour collecter les données de l'utilisateur,
- présentation de données : pour rendre un résultat visible à l'utilisateur,
- *Monitoring* : pour permettre de visualiser l'état de l'application ce qui permet d'observer les performances de l'application,
- *Command-and-Control* : pour réaliser les tâches interactives avec l'application,
- hybride : pour mélanger deux des précédentes classes au moins.

A chacune de ces catégories est associée un *workflow* d'IHM, représenté par un diagramme de séquences, qui place l'IHM en position centrale. La figure 3.16 exprime le *workflow* de l'IHM avec le profil associé en listing 3.21.

```
<Sequence>
  <Action Type="Envoyer" Data="Nom">EnvoyerNom</Action>
  <Action Type="Recevoir" Data="Info">ObtenirInfo</Action>
</Sequence>
```

Listing 3.21 – Profile du *workflow* de l'IHM pour l'obtention des infos d'un employé

Le profil des composants visuels de l'IHM décrit l'ensemble des éléments qui composent l'IHM de manière informelle. Les informations sont ainsi regroupées par "catégories". Ainsi, il est possible de décrire que l'IHM est composée d'une fenêtre avec un titre, de labels, de champs de saisie... Dans l'IHM présente en figure 3.17, les composants visuels sont :

1. Sept *labels* qui désignent un champ de texte, des *labels* et des listes pour afficher le résultat,
2. Un champ de saisie,
3. Trois *labels* pour le résultat,
4. Trois listes pour le résultat,
5. Un bouton *getBusinessInfo*

Le profil de données exprime la capacité de l'IHM en terme de manipulation et d'échange de données. Ce profil comprend quatre catégories qui sont :

- les données entrées qui expriment la capacité à recevoir une information de l'utilisateur,
- les données sorties qui expriment la capacité à afficher un résultat à l'utilisateur,
- l'émission de données qui correspond aux données que l'IHM peut transmettre ainsi que les pré-traitements nécessaires,

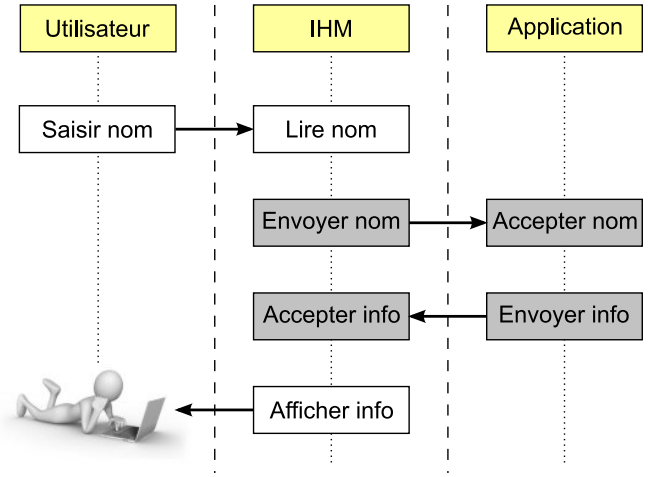


FIGURE 3.16 – Diagramme de séquence représentant le *workflow* de l'IHM pour obtenir les informations d'un employé

FIGURE 3.17 – IHM pour l'obtention des informations d'un employé

– la réception de données qui correspond aux données qui peuvent être reçues par l'IHM de l'application.

L'ontologie associée à l'IHM est complétée par le profil des composants visuels ainsi que par le profil de données qui comprend également les types de données manipulées.

La composition des IHM s'appuie sur l'utilisation des trois éléments suivants :

– un registre de modèle d'applications. Ce modèle d'application contient les spécifications de celles-ci. Il contient également les points de composition d'IHM qui correspondent aux endroits où l'application doit interagir avec l'utilisateur. Le modèle d'application s'appuie sur une représentation proche du diagramme d'activité.
– un registre de services d'IHM qui permet de fournir les IHM. Il donne également la possibilité de réaliser de la découverte et du *matching* d'IHM. Il publie les profils d'IHM décrit auparavant.
– un service de composition d'IHM qui utilise, à la fois, le registre de modèle d'application

et le registre de services d'IHM. Ce service fait également appel à un agent qui recherche l'IHM qui correspond au *workflow* décrit au sein du modèle d'application.

L'ensemble de ces éléments est intégré au sein d'un *framework* automatisant ainsi le processus de composition. A partir d'un modèle d'application, le framework : (i) extrait les points de composition d'IHM, (ii) recherche les services d'IHM correspondants, (iii) compose les services d'IHM au sein du modèle d'applications, (iv) génère et déploie les IHM pour utiliser l'application.

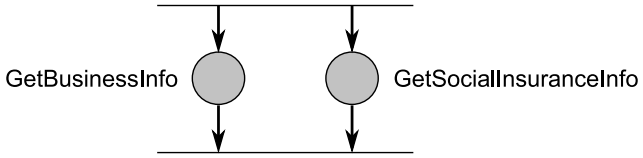


FIGURE 3.18 – Modèle d'application correspond à l'obtention des informations des deux services

Pour conclure sur cette approche, il est **nécessaire de concevoir les IHM en suivant une démarche particulière** qui impose de fournir un grand nombre d'informations telles : le *workflow*, la catégorie de l'IHM, les éléments manipulés... afin de pouvoir les composer. La composition réalisée de l'IHM **ne correspond pas à une composition mais plutôt en la création d'une IHM** dans le sens où **les éléments d'IHM ne peuvent pas être fusionnés**. De plus, cette démarche **impose la spécification de *workflows* spécifiques** intégrant les points de composition d'IHM. Qui plus est, ces *workflows* ne décrivent que l'enchaînement des opérations sans prendre en considération l'usage qui est fait des données.

Fonctionnalités volatiles

Les fonctionnalités volatiles sont des fonctionnalités temporaires que l'on retrouve, par exemple, dans les sites marchands pour des occasions particulières (*e.g.* Noël, Saint Valentin...). Lors de ces événements, le site a besoin de faire évoluer son contenu pour offrir des promotions. Il est donc nécessaire de modifier les pages web. Or, ces modifications doivent être réalisées sur de courtes périodes et ensuite le site doit reprendre son apparence habituelle. Afin de faciliter l'intégration de ces fonctionnalités volatiles au sein du site web, [GRUD07] propose d'utiliser la programmation orientée aspects pour permettre de tisser l'ensemble de la fonctionnalité volatile, c'est-à-dire, la partie fonctionnelle, IHM et navigationnelle aux pages courantes.

Afin de permettre cette composition, il est nécessaire de décrire l'ensemble des fonctionnalités et des IHM associées en suivant le modèle `OOHDM` [SR98]. `OOHDM` se base sur trois modèles pour décrire des applications web : le modèle conceptuel, le modèle de navigation et le modèle d'interface. Le modèle conceptuel représente deux types d'objets : ceux perçus par le modèle de navigation et ceux qui réalisent les calculs, les accès aux bases de données... Le modèle de navigation permet de faire le lien entre les objets du modèle conceptuel et leur représentation dans le modèle d'interface. Il possède également une description de comportement. Dans ce modèle, il existe des contextes de navigation qui permettent de créer des espaces de navigation. Le modèle d'interface s'appuie sur la conception d'interfaces abstraites, et spécifie les objets de l'interface qui sont responsables de la médiation entre une interaction utilisateur et un objet de navigation. La description des interfaces s'appuie sur l'utilisation d'`ADV` (*Abstract Data Views*) [CL95]. Un `ADV` est défini pour chaque nœud présent dans le modèle de navigation. Il permet également de spécifier les interactions et l'effet de ces interactions. Pour ce faire, on utilise les *ADV-Charts* qui correspondent à des diagrammes d'états. On retrouve dans un `ADV` l'ensemble des éléments qui composent l'interface avec leur positionnement et leur type.

Le cœur du site qui correspond à la fonctionnalité principale est décrit en suivant le découpage fourni par `OOHDM`. Les fonctionnalités volatiles suivent ce même découpage. La composition

s'effectue donc entre une fonctionnalité principale et une ou plusieurs fonctionnalités volatiles. Cette composition décrit les liens qui existent entre des éléments du modèle de navigation ainsi que les informations utiles au niveau du modèle conceptuel (*e.g.* dans quel cas la fonctionnalité volatile doit s'appliquer, ce qui dépend d'une valeur présente dans le modèle conceptuel). Pour la partie IHM, la composition correspond au tissage d'un ADV dans un autre ADV. Il s'appuie sur un point de coupe et décrit l'assertion (greffon) qui doit être fait. Les IHM utilisées sont décrites en JSP et la composition au travers de transformations XSLT.

La composition proposée ici, s'appuie sur l'utilisation de la programmation orientée aspect pour permettre le tissage des nouvelles fonctionnalités. L'intérêt est de pouvoir **préserver la fonctionnalité principale et d'y greffer des fonctionnalités temporaires** sans avoir à modifier le code manuellement. Les compositions réalisées ici permettent d'ajouter des éléments dans l'IHM sans pour autant chercher à les fusionner. En effet, toutes les fonctionnalités restent indépendantes ; seules les IHM sont tissées ensemble. On a donc des **juxtapositions d'IHM qui portent une attention particulière sur le placement** des IHM greffées sur l'IHM principale. Par ailleurs, l'utilisation de cette approche implique **un développement spécifique relativement complexe** (au travers d'OOHDM, des ADV, des pages en JSP et des compositions en XSLT).

3.3.3 Synthèse sur la composition d'applications

Les approches de compositions dirigées par les IHM permettent à des utilisateurs finaux de créer leur propre application. C'est ce que l'on retrouve dans les travaux sur l'utilisation des *Mashups* (*i.e. iGoogle*, *Yahoo !Pipe* et [PVM09, Pie09]). Cependant les compositions réalisées permettent seulement de juxtaposer des IHM entre elles ou bien ne donnent pas un contrôle sur le résultat obtenu au niveau du rendu de l'IHM (dans le cas de *Yahoo !Pipe*). L'utilisation des *Web Services* annotés permet d'obtenir un meilleur résultat. Encore une fois, il est possible pour l'utilisateur final de pouvoir créer sa propre application en utilisant les fonctionnalités mises à disposition au travers des *Web Services* [NFPS09]. La description de transferts de données ou bien encore la sélection d'éléments graphiques que l'on souhaite avoir est alors réalisable.

Le développeur peut intervenir pour réaliser la composition d'applications au niveau des IHM en utilisant les *Web Services* annotés qui eux-mêmes utilisent les arbres de tâches [FHSS09]. Cette composition permet de décrire un *workflow* simplifié où seul l'enchaînement des tâches systèmes sont décrites et où il n'est pas possible de spécifier les échanges de données, et donc, la fusion de celles-ci.

Le développeur peut également réaliser la composition d'applications au niveau fonctionnel. Ces compositions imposent un développement spécifique au sein de l'approche proposée. Elles passent, dans un cas, par l'utilisation de la programmation orientée aspects afin de greffer une nouvelle fonctionnalité à une fonctionnalité existante [GRUD07] ou, dans un autre cas, par l'intégration des IHM au sein de SOA [THEC08].

L'ensemble des ces approches s'appuie sur des architectures qui suivent le modèle MVC ; ce dernier rendant possible la distinction entre l'IHM, le noyau fonctionnel et l'interaction qui existe entre les deux. La description qui est faite du noyau fonctionnel ou de l'IHM est uniforme dans chacune des approches dans le sens où les IHM sont homogènes comme les noyaux fonctionnels. Les possibilités de fusion et sélection d'éléments restent limitées et les solutions proposées ne permettent, principalement, que de la juxtaposition d'IHM tout en laissant celles-ci indépendantes. Seules les approches de *Yahoo !Pipe* et [NFPS09] permettent, dans le premier cas, de réaliser des fusions et restrictions sur les données et, dans le second cas, des transferts de données entre les éléments graphiques ainsi que de la sélection d'éléments graphiques.

3.4 Synthèse de l'état de l'art

Cette section établit un bilan des travaux présentés dans les sections précédentes. On extrait de cette synthèse des conclusions et des éléments sur lesquels s'appuie l'approche proposée.

La composition fonctionnelle. L'étude des approches de compositions fonctionnelles fait ressortir deux éléments fondamentaux que sont :

- L'usage des données au travers de la description du flot de données,
- L'usage des opérations au travers de la description du flot de contrôle.

L'utilisation des opérations permet de connaître si l'exécution des opérations s'effectue en parallèle ou bien en séquence et d'introduire la notion de condition. A partir des opérations utilisées, il est possible de savoir quelles sont les données utilisées et comment elles ont été employées.

Que la représentation soit à base de composants [IBM06, Obj08, TLR+09] ou à base des services [Erl05, BHM+04], il est toujours possible d'identifier les opérations qu'ils fournissent et, pour chacune des opérations, les paramètres d'entrée et de sortie qu'ils possèdent. Il est également possible d'identifier les liens qui existent lors de la réalisation des compositions, et ce avec une granularité plus ou moins fine selon l'approche. Les orchestrations de *Web Services* [WS-07] et l'approche WCOMP [HTL+08] permettent par exemple de connaître précisément comment sont utilisées les données au sein d'une composition (données partagées par plusieurs opérations, données fusionnées pour n'en former qu'une seule, ou bien encore résultats utilisés en entrée d'autres opérations). Ces informations sont fort utiles pour identifier la manière dont les éléments sont liés entre eux et sur quels paramètres. Cela permet de savoir le statut des données utiles dans l'application construite par composition : données en entrée, en résultat ou internes.

La composition d'IHM. L'étude de la composition au niveau des IHM fait ressortir essentiellement 2 axes : un axe réutilisation d'IHM existantes par composition et un axe génération d'IHM. Le premier applique des opérateurs de composition sur des éléments qui proviennent de deux IHM en cherchant à traiter les éléments redondants. Les solutions de compositions structurelles [PDF03, LV06, LVM06, LHR+07] d'IHM, basées sur la description de celles-ci, permettent : (i) de supprimer les éléments redondants par application d'une opération d'union, (ii) de regrouper deux IHM entre elles par une opération d'union avec redondance. Par contre, ces solutions ne permettent pas de spécifier un quelconque usage des données (s'il faut les fusionner par exemple dans le cas d'une union). La solution commune à ces travaux est de proposer des descriptions d'IHM abstraites ou concrètes homogènes (*i.e.* réalisées dans le même langage) suffisamment précises pour permettre de retrouver les éléments d'IHM identiques sous peine : (i) de supprimer des éléments qui ne doivent pas l'être ou (ii) de ne pas composer deux éléments qui doivent l'être.

Les constructions d'IHM s'appuient sur plusieurs IHM et cherchent à les regrouper au sein d'une même IHM [BB08]. Les solutions actuelles autour de la composition d'arbres de tâches [LLB07, BB08] se placent en amont dans le processus de développement des IHM ; les arbres de taches n'étant pas embarqués dans les applications une fois construites Elles impliquent donc de générer les IHM correspondantes (suivant le processus de CRF). Ces approches ne tirent, le plus souvent, pas partie de l'utilisation de tâches systèmes, présentes au sein des arbres de tâches : seules les tâches interactives sont utilisées. Cependant l'application d'opération d'union des tâches [LLB07] permet de supprimer les tâches redondantes et, par conséquent, de pouvoir spécifier les données qui seront partagées par plusieurs tâches systèmes sans permettre la fusion des données en provenance de tâches systèmes.

Le niveau de description est soit la vue plus ou moins abstraite soit les tâches utilisateurs. Les deux sont complémentaires la première décrivant plus l'aspect structurel de l'IHM et l'autre les flots de données.

La composition d'applications. Les deux types d'approches offertes pour réaliser la composition d'applications ; dirigée par la composition d'IHM ou par la composition fonctionnelle s'appuient sur des applications qui suivent le découpage suivant :

- un Noyau Fonctionnel (NF) contenant la logique métier de l'application qui comprend l'ensemble des données manipulées ainsi que la manière de les manipuler et l'ensemble des opérations avec lesquelles il est possible d'interagir. Il correspond à un *Web Service* ou à un composant métier.
- une IHM qui fournit un moyen à l'utilisateur d'interagir avec le noyau fonctionnel. Elle est constituée d'éléments graphiques qui vont permettre : (i) de fournir des données au NF, (ii) de recevoir des données en provenance du NF et (iii) de déclencher l'appel d'une opération présente au sein du NF.
- et des liens d'interaction qui permettent de faire le pont entre ces deux éléments. Ces liens décrivent la manière d'utiliser les données et de déclencher les appels aux opérations.

Par contre, l'IHM et les liens d'interaction se retrouvent, plus ou moins, couplés selon les approches, mais restent présents pour permettre de réaliser les compositions. Si certaines approches utilisent des descriptions concrètes des IHM ; les approches de [NFPS09] et de [FHSS09] utilisent des annotations et génèrent les IHM qui n'ont pas été décrites auparavant.

Dans le cadre des compositions dirigées par les IHM, il n'y a aucun impact sur la partie fonctionnelle et seule la partie IHM est composée. Les compositions réalisées restent relativement simples en proposant de juxtaposer deux applications ensemble, de sélectionner des éléments en provenance de deux applications ou bien encore, d'exprimer un transfert de données d'un élément graphique vers un autre [NFPS09]. Il n'est cependant pas possible d'exprimer la fusion de données. Ces compositions permettent donc d'agréger plusieurs IHM tout en laissant indépendantes les parties fonctionnelles des applications impliquées dans la composition. L'approche utilisant les arbres de tâches [FHSS09], pour permettre la réalisation de la composition, n'exploite pas non plus les possibilités offertes par ceux-ci : l'opérateur de composition est présent pour supprimer les éléments redondants sans pour autant vérifier une consistance avec les tâches systèmes.

Dans le cadre des compositions dirigées par la partie fonctionnelle [THEC08, GRUD07], les compositions réalisées n'utilisent pas pleinement les capacités d'expression de celles-ci. Les opérations restent disjointes et aucun échange d'informations n'existe entre elles. Les compositions réalisées au niveau des IHM ne correspondent qu'à des juxtapositions d'IHM sans rechercher à fusionner des éléments. De plus, ces approches nécessitent une mise en œuvre spécifique dans la description des IHM ou des applications.

Constats. Dans ces travaux de thèse, on souhaite pouvoir réutiliser des applications à composer tout en permettant au développeur de tirer partie des outils dont il dispose pour réaliser la composition fonctionnelle. Le but est de proposer une solution qui puisse s'adapter aux différentes technologies existantes pour décrire des applications et qui tire pleinement partie de la composition fonctionnelle pour déduire les éléments d'IHM à conserver. C'est pourquoi, de ces travaux, nous retenons des critères qui nous paraissent indispensables à respecter pour offrir la possibilité de construire des applications interactives par composition en réutilisant au maximum l'existant. Ces critères sont des hypothèses raisonnables qui ont fait leur preuve dans nombre de travaux existants :

1. de construire les applications interactives composables en respectant les architectures qui permettent de distinguer le noyau fonctionnel, de l'IHM et des interactions qui existent.
2. de découper la partie métier en plusieurs éléments afin de faciliter la réutilisation et la composition de ceux-ci pour former des éléments de plus haut niveau.
3. de décrire les échanges entre les différents éléments que ce soit au travers du flot de contrôle seul ou bien couplé avec le flot de données.

4. d'utiliser un langage de description d'interfaces pour décrire la partie structurelle d'une IHM.
5. de respecter l'approche modèle de tâches afin de décrire au mieux une partie de l'interaction entre le noyau fonctionnel et l'IHM.
6. d'exploiter au mieux les informations contenues dans les compositions fonctionnelles pour permettre au niveau IHM de combiner les données se trouvant dans les éléments à unifier et d'avoir la certitude que ces éléments devaient être unis. L'intérêt d'utiliser la composition fonctionnelle, comme point de départ de la composition d'IHM, est de pouvoir gérer la cohérence globale de l'application ainsi que d'exprimer des fusions et des partages de données.
7. d'utiliser des opérations de composition pour faciliter les choix lorsque plusieurs éléments d'IHM sont en concurrence pour des opérations.

Nous déduisons de cet état de l'art et des critères ci-dessus que les applications construites par composition à partir de composants logiciels et/ou de services nous apportent les informations utiles à la construction d'applications interactives si on peut leur associer une description abstraite d'IHM permettant la composition structurelle en explicitant l'interaction avec les fonctionnalités offertes. Aussi, dans les travaux de thèse présentés dans les chapitres suivants, seront décrits l'architecture logicielle des applications composables et le processus de composition mis en œuvre. Le processus s'appuie sur une méta-modélisation des applications permettant de s'abstraire des considérations liées aux plateformes technologiques choisies.

Des raffinements de modèles par utilisation de transformations seront utilisés pour démontrer que ces travaux peuvent être appliqués dans le cadre précis de certaines technologies telles que les *Web Services* et les orchestrations BPEL ainsi que des IHM de type RIA décrites en `Flex` ou XAML. Ces transformations permettront également d'atteindre l'IHM finale. Le processus de composition s'appuiera sur les opérations de compositions utilisées au niveau de la composition fonctionnelle et d'IHM (union ou sélection) ainsi que des possibilités offertes dans la description du flot de données

4

Processus de composition d'applications dirigée par la composition fonctionnelle

Sommaire

4.1 Démarche générale du processus de composition 82
4.1.1 Les données d'entrée du processus de composition 82
4.1.2 Réalisation de la composition d'applications 83
4.1.3 Concrétisation du résultat obtenu 83
4.1.4 Conclusion : mise en œuvre de la démarche 83
4.2 Formalisation des entrées et du résultat de la composition 84
4.2.1 Abstraction d'une application 85
4.2.2 La composition fonctionnelle 99
4.2.3 La composition d'IHM 109
4.2.4 Synthèse et apport de la formalisation et méta-modélisation 116
4.3 Conclusion 117

Ce chapitre présente la démarche que nous avons définie et mise en œuvre pour supporter une composition d'applications dirigée par la composition fonctionnelle. Ce processus de composition repose sur deux points. Le premier est une formalisation du processus de déduction de la composition des IHM et des interactions IHM–noyau fonctionnel à partir de l'analyse des applications à composer et de la composition fonctionnelle effectuée. Le second est une méta-modélisation des applications à composer respectant une architecture précise et de leur composition fonctionnelle. La description de l'architecture des applications (cf. chapitre 3) que l'on souhaite composer et la description de la composition fonctionnelle (cf. section 3.1) permettent d'extraire de l'existant les informations concernant l'usage qui est fait des opérations et des données présentes dans les noyaux fonctionnels. La démarche globale et la formalisation ont été présentées dans [JCDPR11] et une version préliminaire de ces travaux a été présentée dans [CJL09].

L'ensemble de ces informations collectées au niveau des applications à composer et de la composition fonctionnelle est utilisé par le processus de composition pour déduire les éléments graphiques des IHM composées qui doivent être conservés ou fusionnés. Des conflits peuvent apparaître lors de la composition si plusieurs éléments correspondant à une même fonctionnalité sont

sélectionnés (lorsque des paramètres d'entrée ou de sortie de la composition fonctionnelle ont été fusionnés). La résolution des conflits ne peut pas toujours être automatisée et peut nécessiter l'intervention du développeur afin de les résoudre. Le formalisme permet d'identifier clairement comment extraire de l'existant les éléments d'IHM à conserver, les cas de conflits et les propriétés que l'application résultante doit respecter pour assurer la cohérence de l'usage par rapport aux applications à composer.

La section 4.1 présente le processus global de composition. La section 4.2 présente sa formalisation et sa méta-modélisation.

4.1 Démarche générale du processus de composition

Le processus de composition se décompose en quatre étapes détaillées ci-après, qui sont : (i) l'abstraction des applications à composer, (ii) la description de la composition fonctionnelle, (iii) la réalisation de la composition d'applications et (iv) la concrétisation du résultat obtenu.

4.1.1 Les données d'entrée du processus de composition

Deux entrées sont nécessaires à la réalisation de la composition : les applications à composer et la composition fonctionnelle. Les sous-sections qui suivent présentent les descriptions des applications et de la composition fonctionnelle. Ces descriptions sont obtenues par abstraction des applications existantes ainsi que de l'abstraction de la composition fonctionnelle. Ces transformations d'abstraction sont présentées dans le chapitre 5.

Description des applications à composer

La description des applications doit comporter les informations nécessaires à la réalisation du processus de composition d'applications. L'intérêt de cette abstraction est de : (i) se focaliser sur les éléments à composer et (ii) permettre la réutilisation d'applications existantes (par abstraction de celles-ci au sein de cette description). Les informations importantes sont (cf. section 4.2.1) : pour la partie IHM, l'ensemble des éléments qui interagissent avec la partie fonctionnelle c'est-à-dire, les éléments qui permettent à l'utilisateur de fournir des données, les éléments qui visualisent des résultats et les éléments qui déclenchent l'appel d'opérations de la partie fonctionnelle ; pour la partie fonctionnelle, l'ensemble des opérations présentes avec leurs paramètres d'entrée et de sortie ; pour la partie interaction entre les deux précédentes parties, les liens qui décrivent le déclenchement d'appel d'opérations et les liens d'échanges de données. Ceci s'appuie sur les éléments que l'on a pu voir dans le chapitre 3. En effet, c'est ce que l'on retrouve dans les descriptions de services et de composants pour la partie fonctionnelle (cf. section 3.1), dans les descriptions d'IHM Abstraites pour la partie IHM (cf. section 3.2.1). Pour la partie interaction, ce sont les éléments que l'on peut retrouver dans les approches d'assemblage de composants (cf. section 3.1.2) ou bien dans les mashups (cf. section 3.3.1) ou encore dans les approches de compositions d'applications dirigées par la partie fonctionnelle (cf. section 3.3.2). Ce découpage est à la base des architectures MVC, PAC ou Arch sur lesquelles s'appuie l'ensemble des approches de composition d'IHM.

Description de la composition fonctionnelle

La composition fonctionnelle (cf. section 4.2.2) décrit : (i) comment doivent s'enchaîner les opérations et (ii) quel usage est fait des données au sein de la composition. La description de l'enchaînement des opérations (flot de contrôle) spécifie si l'exécution se déroule en parallèle ou en séquence ou bien encore si l'exécution est conditionnée par le résultat d'une précédente opération.

Les flots de données spécifient les échanges de données entre les différentes opérations composées. Ainsi, il est possible d'exprimer qu'une même donnée est utilisée dans plusieurs opérations, qu'une sortie d'une opération est utilisée en entrée d'une autre, ou encore que plusieurs sorties sont fusionnées, c'est-à-dire que leurs des valeurs sont combinées pour n'en former plus qu'une, ce qui peut passer par l'utilisation d'une liste. La description du flot de données et de contrôle s'inspire plus particulièrement des orchestrations de *Web Services* (cf. section 3.1.1) qui spécifient de manière plus précise les interactions et les échanges de données entre les opérations. Point que l'on retrouve également dans l'approche de WCOMP mais que l'on perd dans les approches à composants masque cette information au sein de l'implémentation du composant (cf. section 3.1.2).

4.1.2 Réalisation de la composition d'applications

La composition s'appuie sur l'ensemble des éléments précédemment décrits.L'objectif est alors de déduire les éléments d'IHM qui doivent être conservés, fusionnés et de créer les liens entre ces éléments d'IHM et la composition fonctionnelle afin de fournir une IHM qui utilise la composition fonctionnelle. Le résultat du processus de composition est donc l'obtention d'une nouvelle application qui réutilise des éléments présents dans les IHM existantes. Le processus de sélection des éléments d'IHM qui doivent être conservés peut faire intervenir un développeur afin de résoudre un conflit. Ces conflits peuvent apparaître lors de fusion au niveau de la partie fonctionnelle. A ce moment là, plusieurs éléments d'IHM peuvent être sélectionnés. Dans le cas où ils sont différents, le développeur doit faire un choix pour savoir si l'on conserve l'ensemble des éléments ou bien un seul élément ou encore si l'on crée un nouvel élément pour l'IHM résultante. Ce point est une des originalités de ce travail. En effet, l'état de l'art nous montre que les conflits ne sont pris en considération ni dans les travaux sur la composition d'IHM (cf. section 3.2), ni dans la composition d'application (cf. section 3.3). Dans la composition d'IHM, seuls les éléments qui sont considérés comme égaux sont fusionnés (*i.e.* qu'un seul élément est conservé). Dans le cas de la composition d'applications, les compositions fonctionnelles simples n'introduisent pas de fusion d'éléments d'IHM. Cette IHM résultante correspond à un "squelette" d'IHM qui servira de base de travail au développeur afin d'être enrichie pour obtenir une IHM destinée à un utilisateur final.

4.1.3 Concrétisation du résultat obtenu

La concrétisation vise l'obtention d'un rendu visuel de l'IHM résultante du processus de composition (cf. section 5.1.2). Ce rendu visuel permet au développeur de se rendre compte des éléments qui ont été conservés et de vérifier qu'ils correspondent bien au résultat souhaité. Cette concrétisation transforme ainsi la description abstraite de l'IHM en une description concrète dans un langage cible. Par ailleurs, cette concrétisation peut être enrichie par des informations de placement ou bien encore en ajoutant des éléments d'IHM supplémentaires afin d'obtenir une IHM utilisable par un utilisateur final.

4.1.4 Conclusion : mise en œuvre de la démarche

Ces étapes sont reprises au sein de la figure 4.1 qui illustre la mise en œuvre du processus de composition d'applications. On retrouve ainsi les étapes d'abstraction des applications (point 1), d'abstraction de la composition fonctionnelle (point 2), de la réalisation de la composition d'IHM (point 3) et de la concrétisation du résultat obtenu (point 4). Cette figure reprend l'ensemble des éléments d'entrée en spécifiant les différentes technologies qui peuvent intervenir à ces étapes et le résultat obtenu de la composition.

La section 4.2 présente la description la formalisation et la méta-modélisation de l'ensemble des éléments d'entrée et de sortie du processus de composition. Ces deux descriptions repré-

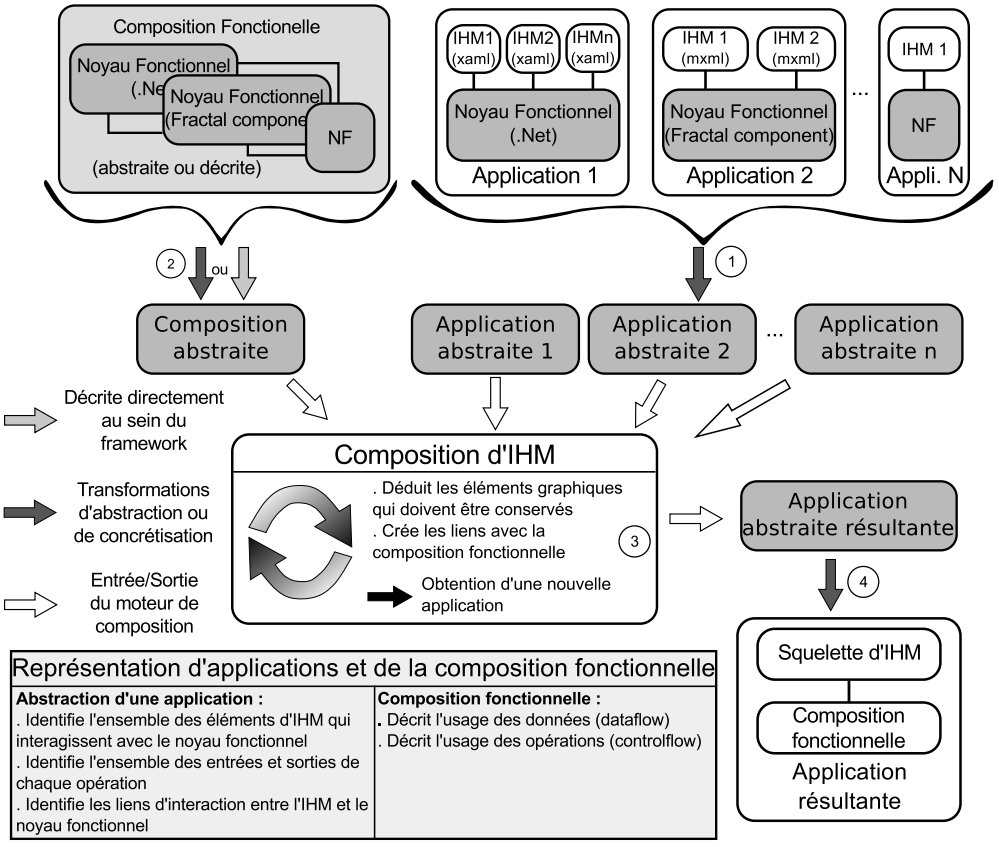


FIGURE 4.1 – Démarche globale pour la réalisation de la composition d'applications

sentent la même information, mais sont utilisées à des niveaux différents. La formalisation est utilisée au sein des algorithmes de composition d'IHM tandis que la méta-modélisation est utilisée au sein des transformations d'abstraction et de concrétisation.

4.2 Formalisation des entrées et du résultat du processus de composition

Cette section présente la formalisation et la méta-modélisation de l'ensemble des informations nécessaires à la réalisation du processus de composition ainsi que le résultat de celui-ci. L'intérêt de ces deux représentations réside dans le fait qu'elles sont utilisées à deux niveaux du processus de composition. En effet, la formalisation s'appuie sur la théorie des ensembles, est utilisée au sein des algorithmes de composition et elle permet d'identifier des propriétés sur ceux-ci. La méta-modèlisation est utilisée par les transformations nécessaires à l'abstraction des applications et de la composition fonctionnelle ainsi qu'à la concrétisation du résultat de la composition d'IHM. Le noyau fonctionnel, les IHM ainsi que la composition fonctionnelle sont représentés par des composants qui possèdent des ports de données (en entrée et en sortie) et d'événement/action. Le méta-modèle utilisé est `AliasComponent`. Il est présenté dans son intégralité en figure 4.2. Tout au long des sections qui suivent, il est présenté par partie afin de faciliter sa compréhension et également sa lecture.

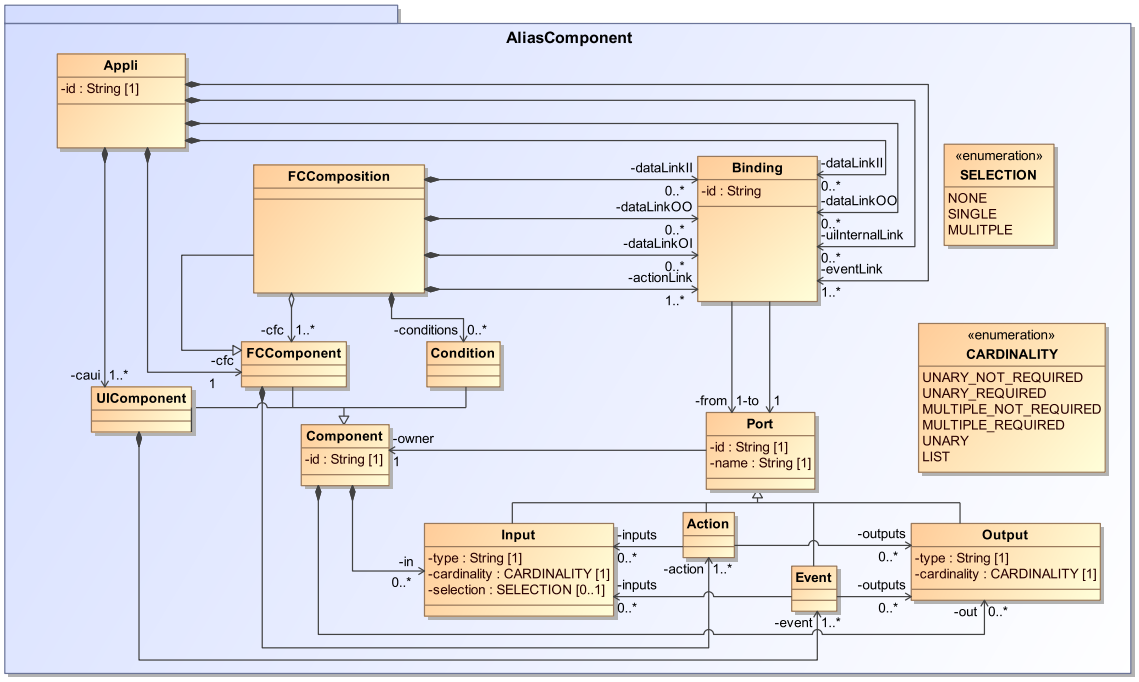


FIGURE 4.2 – Méta-modèle `AliasComponent`dans son intégralité

Pour résumer et avant de détailler chacun des éléments qui composent le méta-modèle `AliasComponent`, celui-ci permet de décrire une application (*Appli*), comme un assemblage de composants : fonctionnel (*cfc* de type *FCComponent*) et d'IHM (*caui* de type *UIComponent*). Ces deux types de composants sont liés entre eux par des liens (*Binding*) de données et d'événements. Chaque composant est constitué de port de données en entrée (*in* de type *Input*) et en sortie (*out* de type *Output*). Dans le cas d'un composant d'IHM, on a un port d'événement (*event* de type *Event*) et dans le cas d'un composant fonctionnel on a un port d'action correspond à l'opération (*action* de type *Action*). Enfin un composant fonctionnel peut-être un composant composite (*FCComposition*).

La sous-section 4.2.1 présente les données qui caractérisent une application au niveau fonctionnel, IHM et interaction.

La sous-section 4.2.2 présente les données qui décrivent la composition fonctionnelle. La composition est représentée par un composant composite qui agrège des sous-composants et propose des regroupements de fonctionnalités ou des fonctionnalités de plus haut-niveau.

La sous-section 4.2.3 spécifie le résultat de la composition d'IHM que l'on doit obtenir ainsi que les propriétés qu'il doit avoir. Elle décrit également l'ensemble des étapes pour réaliser la composition.

4.2.1 Abstraction d'une application

Une application se décompose en trois parties : le noyau fonctionnel, des IHM et les interactions entre les deux parties précédentes. Ce qui suit décrit chacune de ces parties de manière formelle et au sein du méta-modèle. Chacune des parties est illustrée sur l'exemple de l'application *Business Application* (cf. 2.1.1). Cette application comporte en partie fonctionnelle, un *Web Service* qui dispose de deux opérations (une opération qui permet d'obtenir l'ensemble des informations d'un employé à partir de son nom et une opération qui ne retourne que les adresses

associées à un employé) et en partie IHM, deux interfaces pour utiliser les deux opérations proposées.

Une application peut posséder plusieurs IHM qui correspondent aux actions présentes dans le composant fonctionnel. Il existe deux types de liens d'interaction : les liens de données qui décrivent les échanges de données entre l'IHM et le fonctionnel, et les liens d'événement qui décrivent le déclenchement de l'appel d'une opération. Les liens de données relient soit les entrées entre elles (une entrée du composant d'IHM vers une entrée du composant fonctionnel) soit les sorties entre elles (une sortie du composant fonctionnel vers une sortie du composant d'IHM). Les liens d'événements relient un événement du composant d'IHM avec une action du composant fonctionnel.

Formalisation. La définition 1 correspond à la formalisation de l'abstraction d'une application qui se caractérise par un identifiant, un composant fonctionnel, des composants d'IHM et un ensemble de liens d'interactions. Dans la suite, on utilise la notion de *String* dans les fonctions. On appelle *String* n'importe quelle chaîne de caractères non vide.

Définition 1 (Application)**.** Une application abstraite ($appli$) est définie par :
- un identifiant unique (*id*) obtenu par la fonction : $APP\_ID : APPLI \rightarrow String, app\_id(appli) = id$,
- un composant fonctionnel (cfc) correspondant au noyau fonctionnel de l'application, c'est-à-dire, l'abstraction du *Web Service* ou du composant métier. Il est dénoté : $APP\_CFC : APPLI \rightarrow CFC, app\_cfc(appli) = cfc$,
- un ensemble de composants d'IHM ($\{caui_1, \ldots, caui_n\}$) correspondant aux IHM associées au noyau fonctionnel. Les composants d'IHM correspondent à l'abstraction des IHM existantes. C'est ensemble est obtenu par : $APP\_CAUI : APPLI \rightarrow CAUI^+, app\_caui(appli) = \{caui_1, \ldots, caui_n\}$,
- des liens d'interaction entre le composant fonctionnel et un composant d'IHM qui décrivent les échanges de données et le déclenchement d'événements définis par les fonctions :
  - $DataLinkII$, ils correspondent aux liens d'échange de données entre deux entrées qui relient un élément graphique qui permet à l'utilisateur de saisir une information à une entrée du composant fonctionnel,
  - $DataLinkOO$, ils correspondent aux liens d'échange de données entre deux sorties qui relient un élément graphique qui permet l'affichage d'un résultat du composant fonctionnel,
  - $EventLink$, ils correspondent aux liens entre un événement d'un composant d'IHM qui déclenchent l'appel d'une action du composant fonctionnel.

L'ensemble des liens qui sont décrits ici, est caractérisé par des *Binding* au sein du méta-modèle que l'application possède. On retrouve par ailleurs les deux types de composants : le composant fonctionnel décrit par *FCComponent* et les composants d'IHM décrits par *UIComponent*.

Méta-modélisation. La figure 4.3 représente le méta-modèle `AliasComponent` de l'application. On retrouve la définition du composant fonctionnel et des composants d'IHM. On voit qu'une application possède un composant fonctionnel et peut posséder un ou plusieurs composants d'IHM. Les ports des composants sont reliés entre eux au travers de *Bindings*. Les *Bindings* font référence à un port source (*from*) et un port de destination (*to*). L'application possède trois types de liens qui sont : *dataLinkII*, *dataLinkOO* et *eventLink*. Ces listes de liens reprennent les types de liens présents dans la définition 1.

Ce qui suit détaille dans un premier temps ce qu'est un composant fonctionnel puis un composant d'IHM et enfin finit par la description des liens d'interaction.

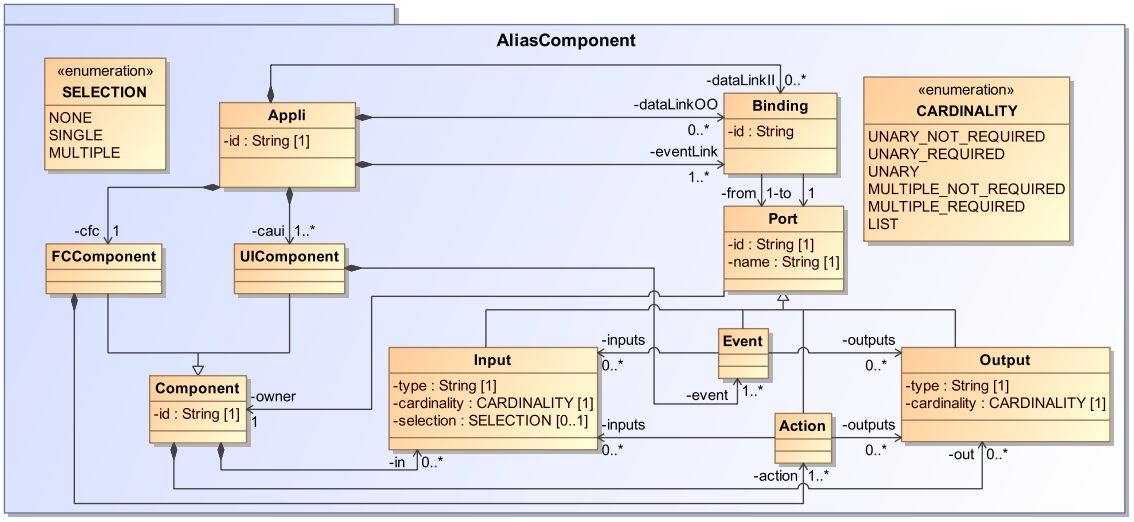


FIGURE 4.3 – Méta-modèle `AliasComponent`de l'application

Composant fonctionnel

L'abstraction du Noyau Fonctionnel (NF) permet aussi bien de représenter un service (*e.g.* *Web Service*) qu'un composant (*e.g.* les composants FRACTAL, WCOMP). Cette abstraction décrit la partie fonctionnelle de l'application sous la forme d'un composant. Ce composant possède deux types de ports qui sont : (i) des ports de données et (ii) des ports d'action. Les ports d'action correspondent aux opérations qui sont disponibles au sein de la partie fonctionnelle. A chaque port d'action sont associés des ports de données. Ces ports de données peuvent être des ports d'entrée qui correspondent aux paramètres à fournir pour exécuter une opération ou des ports de sorties qui correspondent aux retours d'exécution de l'opération. Le port d'action associé à ses ports d'entrée/sortie forme le prototype de l'opération. Cette représentation permet de couvrir l'ensemble des descriptions que l'on veut avoir du noyau fonctionnel (cf. section 3.1). L'ensemble de ces informations forme l'abstraction de la partie fonctionnelle.

Formalisation. Le composant fonctionnel est formalisé par la définition 2. Il comprend un ensemble de ports de données en entrée et sortie et des ports d'action correspondant aux opérations.

Définition 2 (FCComponent)**.** Le composant fonctionnel (cfc) d'une application ($appli$) est obtenu par la fonction : $app\_cfc(appli)$. Il est défini par :

- un identifiant unique (id) obtenu par la fonction : $CFC\_ID : CFC \rightarrow String, cfc\_id(cfc) = id$
- un ensemble d'*action*. L'ensemble des *action* est obtenu par la fonction : $CFC\_ACTIONS : CFC \rightarrow ACTION^{+}, cfc\_actions(cfc) = \{action_1, \ldots, action_n\}$. Chaque *action* est définie par :
  - un identifiant unique id obtenu par la fonction : $ACTION\_ID : ACTION \rightarrow String$, $action\_id(action) = id$
  - un nom ($name$) qui correspond au nom de l'opération, obtenu par la fonction : $ACTION\_NAME : ACTION \rightarrow String, action\_name(action) = name$
  - un ensemble d'entrées ($inputs$) obtenu par la fonction : $ACTION\_INPUTS : ACTION \rightarrow INPUT^{*}, action\_inputs(action) = inputs$ et un ensemble de sorties ($outputs$) obtenu par la fonction : $ACTION\_OUTPUTS : ACTION \rightarrow OUTPUT^{*}, action\_outputs(action)$. Ces deux ensembles correspondent aux paramètres d'entrée et de sortie de l'opération.

– deux fonctions donnent les entrées et les sorties du composant fonctionnel :

$$CFC\_INPUTS : CFC \rightarrow INPUT^* \tag{4.1}$$

$$cfc\_inputs(cfc) = \left\{ \begin{array}{l} i \mid i \in action\_inputs(action), \\ \forall action \in cfc\_actions(cfc) \end{array} \right\} \tag{4.2}$$

$$CFC\_OUTPUTS : CFC \rightarrow OUTPUT^* \tag{4.3}$$

$$cfc\_outputs(cfc) = \left\{ \begin{array}{l} o \mid o \in action\_outputs(action), \\ \forall action \in cfc\_actions(cfc) \end{array} \right\} \tag{4.4}$$

Les entrées et sorties possèdent les mêmes caractéristiques qui sont :
- un identifiant unique (*id*) obtenu par les fonctions : $CFC\_IN\_ID : INPUT \rightarrow String$, $cfc\_in\_id(in) = id$ ou $CFC\_OUT\_ID : OUTPUT \rightarrow String, cfc\_out\_id(out) = id$
- un nom (*name*) correspondant au nom du paramètre. Ce nom est obtenu par les fonctions : $CFC\_IN\_NAME : INPUT \rightarrow String, cfc\_in\_name(in) = name$ ou $CFC\_OUT\_NAME : OUTPUT \rightarrow String, cfc\_out\_name(out) = name$
- un *type* qui peut être un type primitif (entier, chaîne de caractères, flottant...) ou un type complexe (basé sur un ensemble de types primitifs). Le type est obtenu par les fonctions : $CFC\_IN\_TYPE : INPUT \rightarrow String, cfc\_in\_type(in)$ ou $CFC\_OUT\_TYPE : OUTPUT \rightarrow String, cfc\_out\_type(out)$. On utilise ici une chaîne de caractères pour spécifier le type pour permettre de spécifier que c'est un type complexe.
- une cardinalité (*cardinality*) qui permet de décrire si l'élément est requis ou non, et s'il correspond à un scalaire ou une liste. La cardinalité est obtenue par les fonctions : $CFC\_IN\_CARDINALITY : INPUT \rightarrow CARDINALITY, cfc\_in\_cardinality(in) = cardinality \mid cardinality \in \{[0,1],[0,n],[1,1],[1,n]\}$ ou $CFC\_OUT\_CARDINALITY : OUTPUT \rightarrow CARDINALITY, cfc\_out\_cardinality(out) = cardinality \mid cardinality \in \{[0,1],[0,n],[1,1],[1,n]\}$. La cardinalité est définie par un couple dont les possibilités sont les suivantes : $\{[0,1],[0,n],[1,1],[1,n]\}$. La première valeur permet de spécifier si l'élément est optionnel ou obligatoire et peut prendre pour valeur : 0 (optionnel) ou 1 (obligatoire). La seconde valeur permet de spécifier si l'élément correspond à une liste ou pas et peut prendre pour valeur : 1 (valeur unique) ou n (liste de valeurs).

Dans le cas de la description d'un *Web Service*, chaque élément décrit dans le schéma de données peut posséder deux caractéristiques : *minOccurs* et *maxOccurs* qui correspondent aux caractéristiques que l'on retrouve dans la cardinalité.

Un composant fonctionnel peut également être un composant composite. Dans ce cas, il est connecté à ses sous-composants au travers de liens et il respecte les propriétés décrites dans la définition 7 en plus de celles présentes dans la définition 2.

Les deux fonctions $cfc\_inputs(cfc)$ et $cfc\_outputs(cfc)$ qui permettent d'obtenir l'ensemble des entrées et sorties du composant fonctionnel correspondent aux éléments *in* et *out* au sein du méta-modèle `AliasComponent`. De même les valeurs possibles prises par la cardinalité (*cardinality*) sont décrites au sein d'une énumération. L'équivalence qui est faite entre le formalisme et la représentation au sein du méta-modèle est la suivante :
- $[0,1] \Leftrightarrow$ *UNARY_NOT_REQUIRED*,
- $[0,n] \Leftrightarrow$ *MULTIPLE_NOT_REQUIRED*,
- $[1,1] \Leftrightarrow$ *UNARY_REQUIRED*,
- $[1,n] \Leftrightarrow$ *MULTIPLE_REQUIRED*.

Méta-modélisation. La figure 4.4 présente le composant fonctionnel au sein du méta-modèle `AliasComponent`. On retrouve de la même manière que dans la formalisation le fait qu'un composant fonctionnel (*FCComponent*) possède un ensemble de ports de données en entrée et en sortie

ainsi que des ports d'action qui correspondent aux opérations. Les ports d'action sont obligatoires car ils correspondent aux opérations fournies par le composant fonctionnel. Mais il est possible que ces opérations ne prennent pas paramètre d'entrée et ne retournent aucune valeur. C'est pourquoi les ports d'entrée et de sortie sont optionnels. Enfin un port d'action possède des références vers des ports d'entrée et de sortie ce qui permet de représenter ses paramètres d'entrée et de sortie. Les ports possèdent également une référence sur le composant auxquels ils appartiennent. Les ports de données possèdent un attribut de type *cardinality* et de type *type*.

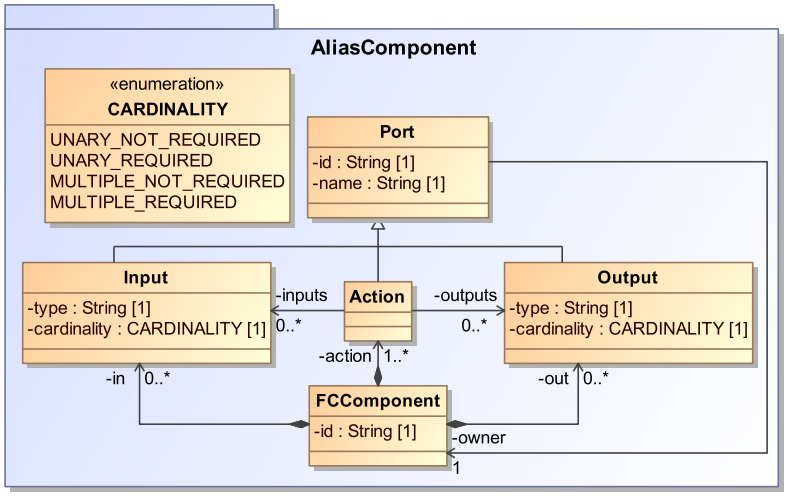


FIGURE 4.4 – Représentation du composant fonctionnel au sein du méta-modèle `AliasComponent`

La cardinalité associée aux entrées et sorties d'un composant fonctionnel doit posséder une valeur présente dans l'énumération qui la caractérise. Cette cardinalité est contrainte par une règle OCL fournie par le listing 4.1. Cette contrainte permet de spécifier les valeurs que l'attribut *cardinality* par rapport aux valeurs qu'il peut prendre. Cette contrainte se retrouve dans la description de l'application et l'énumération se retrouve partagé avec la description du composant d'IHM qui peut prendre d'autres valeurs.

```
1  context FCComponent inv cardinality :
   self.in->forAll(
       ((cardinality = CARDINALITY::UNARY_NOT_REQUIRED) xor
        (cardinality = CARDINALITY::UNARY_REQUIRED) xor
        (cardinality = CARDINALITY::MULTIPLE_NOT_REQUIRED) xor
6       (cardinality = CARDINALITY::MULTIPLE_REQUIRED)))
   and self.out->forAll(
       ((cardinality = CARDINALITY::UNARY_NOT_REQUIRED) xor
        (cardinality = CARDINALITY::UNARY_REQUIRED) xor
        (cardinality = CARDINALITY::MULTIPLE_NOT_REQUIRED) xor
11      (cardinality = CARDINALITY::MULTIPLE_REQUIRED)))
```

Listing 4.1 – Règle OCL qui contraint l'attribut *cardinality* sur les ports de données en entrée et sortie

Exemple. Pour illustrer le formalisme et la méta-modélisation, nous utilisons le *Web Service* décrit en section 2.1.1. Il est composé de deux opérations `getBusinessInfo` et `getAddresses`. Ces deux opérations prennent en paramètre une chaîne de caractères correspondant au nom complet de la personne que l'on recherche et retourne dans le premier cas un élément de type `businessInformation` composé de six éléments et dans le second cas un ensemble d'adresses

(liste de chaînes de caractères). Il est représenté de manière formelle dans l'exemple 1 et au sein du méta-modèle `AliasComponent` dans la figure 4.5.

Exemple 1 (BusinessService). $cfc_{BusinessService} = app\_cfc(BusinessApplication)$ correspond à la formalisation du composant fonctionnel de ce même *Web Service*. Celle-ci correspond à :

$$
\begin{aligned}
cfc\_id(cfc_{BusinessService}) &= business_{fc} \\
cfc\_actions(cfc_{BusinessService}) &= \{action_1, action_2\} \\
action\_id(action_1) &= fc\_action_1 \\
action\_name(action_1) &= getBusinessInfo \\
action\_inputs(action_1) &= \{input_1\} \\
action\_outputs(action_1) &= \left\{ \begin{array}{l} output_1, output_2, output_3, \\ output_4, output_5, output_6 \end{array} \right\} \\
action\_id(action_2) &= fc\_action_2 \\
action\_name(action_2) &= getAddresses \\
action\_inputs(action_2) &= \{input_2\} \\
action\_outputs(action_2) &= \{output_7\}
\end{aligned}
$$

X	$cfc\_in\_id(X)$	$cfc\_in\_name(X)$	$cfc\_in\_type(X)$	$cfc\_in\_cardinality(X)$
$input_1$	fc_input_1	$fullName$	$string$	$[1,1]$
$input_2$	fc_input_2	$fullName$	$string$	$[1,1]$
X	$cfc\_out\_id(X)$	$cfc\_out\_name(X)$	$cfc\_out\_type(X)$	$cfc\_out\_cardinality(X)$
$output_1$	fc_output_1	$address$	$string$	$[0,n]$
$output_2$	fc_output_2	$building$	$string$	$[0,n]$
$output_3$	fc_output_3	$email$	$string$	$[0,1]$
$output_4$	fc_output_4	$fullName$	$string$	$[0,1]$
$output_5$	fc_output_5	$office$	$string$	$[0,n]$
$output_6$	fc_output_6	$position$	$string$	$[0,1]$
$output_7$	fc_output_7	$addresses$	$string$	$[0,n]$

Composant d'IHM

L'abstraction d'une IHM repose sur les mêmes fondements que l'abstraction du NF. En effet, le résultat de cette abstraction correspond à un composant d'IHM. On retrouve des ports de données en entrée et en sortie, ceux-ci permettent de faire la saisie des informations qui seront passées en paramètres d'entrée d'une opération ($ACTION$ du $FCComponent$) et d'afficher le résultat de l'exécution d'une opération ($ACTION$ du $FCComponent$). Dans le cas des entrées, il y a également une notion de sélection qui est prise en considération puisque l'utilisateur peut certaines fois choisir parmi une liste de possibilités. Les ports d'événement de l'abstraction de l'IHM permettent de faire le lien avec les éléments d'entrée et de sortie de la même manière que les actions dans le composant fonctionnel. Les ports d'événements représentent les éléments d'IHM avec lesquels l'utilisateur interagit pour déclencher l'appel d'une opération.

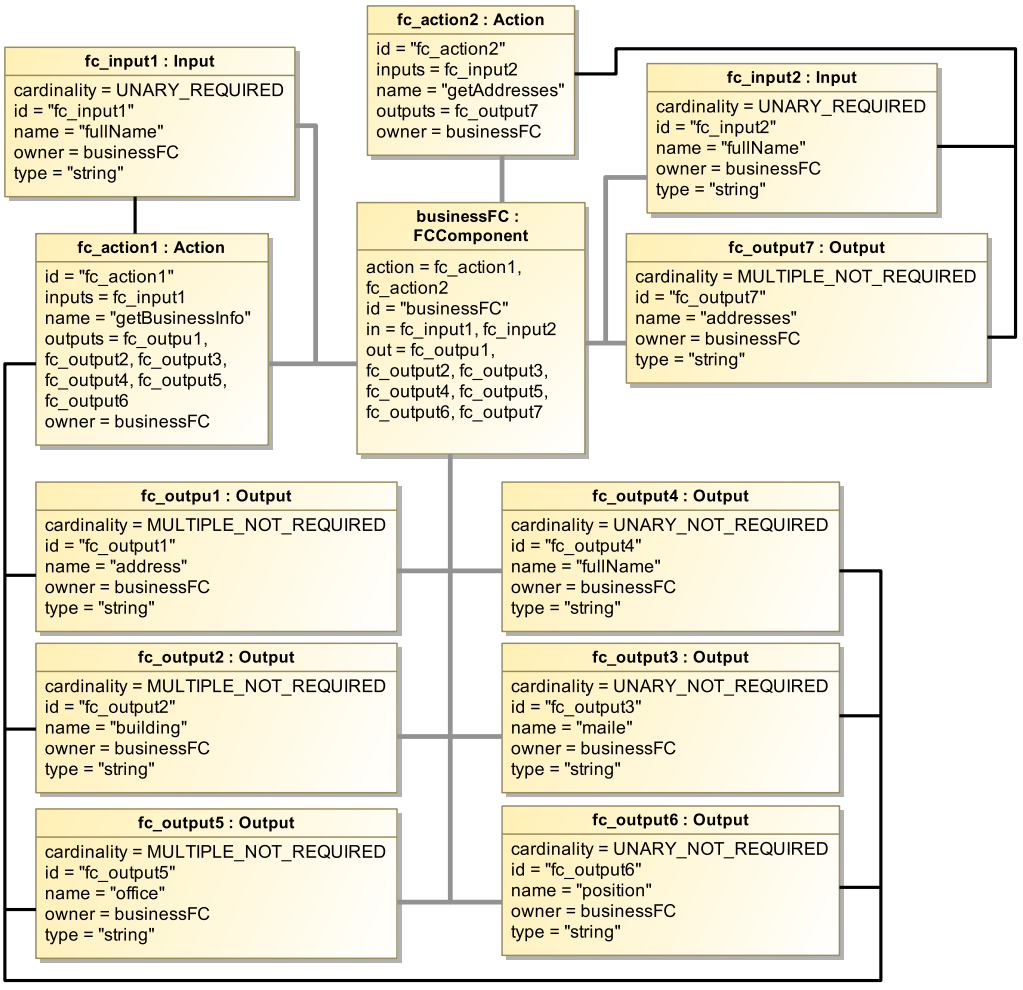


FIGURE 4.5 – Instantiation du service *BusinessService* au sein du méta-modèle `AliasComponent`

Formalisation. L'IHM est formalisée par la définition 3. Elle correspond à un composant qui possède un ensemble de ports de données en entrée et sortie et à des ports d'événement qui permettent de déclencher l'appel d'une opération.

Définition 3 (UIComponent)**.** Le composant d'IHM (*caui*) d'une application (*appli*) appartient à l'ensemble des IHM que l'on obtient par la fonction : $app\_caui(appli)$. Il est défini par :

- un identifiant unique (*id*) obtenu par la fonction : $CAUI\_ID : CAUI \rightarrow String, caui\_id(caui) = id$
- un ensemble d'*event*. L'ensemble des *event* est obtenu par la fonction : $CAUI\_EVENTS : CAUI \rightarrow EVENT^{+}, caui\_events(caui) = \{event_1, \ldots, event_n\}$. Chaque *event* est définie par :
  - un identifiant unique *id* obtenu par la fonction : $EVENT\_ID : EVENT \rightarrow String$, $event\_id(event) = id$
  - un nom (*name*) qui correspond au nom de l'événement (*e.g.* nom du bouton), obtenu par la fonction : $EVENT\_NAME : EVENT \rightarrow String, event\_name(event) = name$

  - un ensemble d'entrées ($inputs$) obtenu par la fonction : $EVENT\_INPUTS : EVENT \rightarrow INPUT^*, event\_inputs(event) = inputs$ et un ensemble de sorties ($outputs$) obtenu par la fonction : $EVENT\_OUTPUTS : EVENT \rightarrow OUTPUT^*, event\_outputs(event) = outputs$. Ces deux ensembles correspondent aux paramètres d'entrée et de sortie de cet événement.
- deux fonctions donnent les entrées et les sorties du composant d'IHM :

$$CAUI\_INPUTS : CAUI \rightarrow INPUT^* \tag{4.5}$$

$$caui\_inputs(caui) = \left\{ \begin{array}{l} i \mid i \in event\_inputs(event), \\ \forall event \in caui\_events(caui) \end{array} \right\} \tag{4.6}$$

$$CAUI\_OUTPUTS : CAUI \rightarrow OUTPUT^* \tag{4.7}$$

$$caui\_outputs(caui) = \left\{ \begin{array}{l} o \mid o \in event\_outputs(event), \\ \forall event \in caui\_events(caui) \end{array} \right\} \tag{4.8}$$

Les entrées (resp. sorties) possèdent les caractéristiques suivantes :

- un identifiant unique (id) obtenu par la fonction : $CAUI\_IN\_ID : INPUT \rightarrow String$, $caui\_in\_id(in) = id$ (resp. $CAUI\_OUT\_ID : OUTPUT \rightarrow String, caui\_out\_id(out) = id$).
- un nom ($name$) correspondant au nom du label qui désigne l'élément d'IHM. Ce nom est obtenu par la fonction : $CAUI\_IN\_NAME : INPUT \rightarrow String, caui\_in\_name(in) = name$ (resp. $CAUI\_OUT\_NAME : OUTPUT \rightarrow String, caui\_out\_name(out) = name$).
- un $uitype$ qui est un sous-ensemble des types primitifs présents dans les descriptions de la partie fonctionnelle. Les valeurs possibles sont : $\{number, string, date, boolean\}$. Ils correspondent aux types gérés par l'élément d'IHM. L'$uitype$ est obtenu par la fonction : $CAUI\_IN\_UITYPE : INPUT \rightarrow String, caui\_in\_uitype(in) = uitype \mid uitype \in \{number, string, date, boolean\}$ (resp. $CAUI\_OUT\_UITYPE : OUTPUT \rightarrow UITYPE, caui\_out\_uitype(out) = uitype \mid uitype \in \{number, string, date, boolean\}$).
- une cardinalité ($cardinality$) qui spécifie si l'on doit fournir (resp. afficher) une valeur unique ou multiple. Une cardinalité égale à 1 signifie que l'élément d'IHM prend (resp. affiche) une unique valeur. Une cardinalité égale à n signifie que l'élément d'IHM prend (resp. affiche) un ensemble de valeurs. La cardinalité est obtenue par la fonction : $CAUI\_IN\_CARDINALITY : INPUT \rightarrow CARDINALITY, caui\_in\_cardinality(in) = cardinality \mid cardinality \in \{1, n\}$ (resp. $CAUI\_OUT\_CARDINALITY : OUTPUT \rightarrow CARDINALITY, caui\_out\_cardinality(out) = cardinality \mid cardinality \in \{1, n\}$).
- Pour une cardinalité égale à n seulement, il est possible d'ajouter une sélection ($selection$) associée à l'élément d'IHM. La sélection spécifie le type de sélection fournie par l'élément d'IHM. Les différentes valeurs prises sont : $\{none, single, multiple\}$. La sélection est obtenue par la fonction : $CAUI\_IN\_SELECTION : INPUT \rightarrow SELECTION, caui\_in\_selection(in) = selection \mid selection \in \{none, single, multiple\}$.

Les deux fonctions $caui\_inputs(caui)$ et $caui\_outputs(caui)$ qui permettent d'obtenir l'ensemble des entrées et sorties du composant fonctionnel correspondent aux éléments in et out au sein du méta-modèle `AliasComponent`. De même les valeurs possibles prises par la cardinalité ($cardinality$) sont décrites au sein d'une énumération. L'équivalence qui est faite entre le formalisme et la représentation avec le méta-modèle est la suivante :

- $1 \Leftrightarrow UNARY$,
- $n \Leftrightarrow LIST$,

De la même manière que pour la cardinalité, une équivalence pour la sélection ($selection$) entre le formalisme et la représentation au sein du méta-modèle. Cette équivalence est directe puisque l'on utilise les mêmes valeurs : $none$, $single$ et $multiple$.

Méta-modélisation. La figure 4.6 représente le composant d'IHM (*UIComponent*) au sein d'`AliasComponent`. On retrouve un ensemble de ports de données en entrée et en sortie ainsi que

des ports d'événement qui correspondent aux éléments qui déclenchent l'appel d'une opération (*i.e.* un port *ACTION* du composant fonctionnel). De la même manière que pour le composant fonctionnel, les ports d'événements sont obligatoires et les ports de données d'entrée et sortie ne le sont pas et dépendent de ceux que requiert le composant fonctionnel. Enfin un port d'événement possède des références vers des ports d'entrée et de sorties ce qui permet d'associer les différents éléments graphiques à un événement. Les ports possèdent également une référence sur le composant auxquels ils appartiennent. Les ports de données possèdent des attributs de type *cardinality*, *uitype* et *selection* dans le cas d'une entrée.

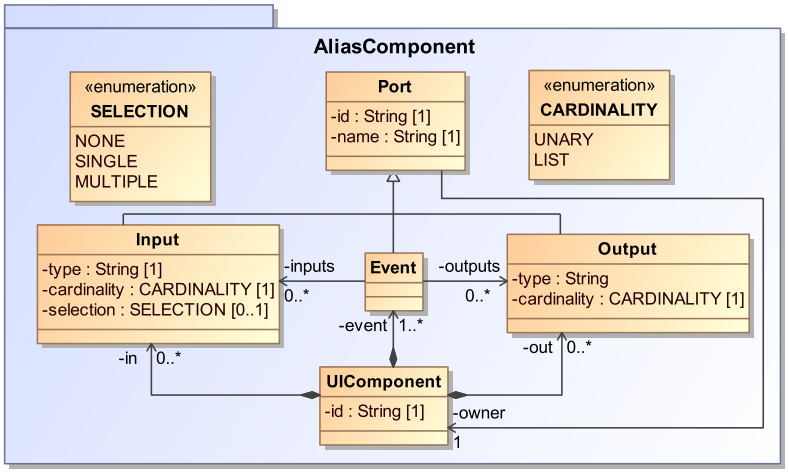


FIGURE 4.6 – Représentation du composant d'IHM au sein du méta-modèle `AliasComponent`

La cardinalité est contrainte par une règle OCL fournie par le listing 4.2.

```
context UIComponent inv ui_cardinality :
self.in->forAll(
    ((cardinality = CARDINALITY::UNARY) xor (cardinality = CARDINALITY::LIST))) and
self.out->forAll(
    ((cardinality = CARDINALITY::UNARY) xor (cardinality = CARDINALITY::LIST)))
```

Listing 4.2 – Règle OCL qui contraint l'attribut *cardinality* sur les ports de données en entrée et sortie

Enfin, un port d'entrée peut posséder une sélection qui vaut : *NONE*, *SINGLE* ou *MULTIPLE*. Cette sélection est contrainte par une règle OCL qui spécifie que seuls les ports d'entrée qui possèdent une cardinalité qui vaut *LIST* doivent avoir une valeur pour sélection sinon, dans le cas contraire, la sélection ne doit pas être définie. Ce qui est défini par la règle du listing 4.3.

```
context UIComponent inv ui_selection :
self.in->forAll(
  if selection.oclIsUndefined() then cardinality = CARDINALITY::UNARY
  else cardinality = CARDINALITY::LIST
  endif)
```

Listing 4.3 – Règle OCL qui contraint l'attribut *selection* sur le port d'entrée

Exemple. Pour illustrer la description du composant d'IHM, nous utilisons une des IHM présentes dans l'application décrite en section 2.1.1 et illustrée par la figure 4.7. Cette IHM possède une entrée pour saisir le nom de la personne dont on souhaite afficher les informations. Il y a un bouton pour déclencher l'appel à l'opération et six sorties permettant d'afficher le résultat de

l'opération `getBusinessInfo`. Cette IHM est décrite de manière formelle dans l'exemple 2 et au sein du méta-modèle `AliasComponent` dans la figure 4.8.

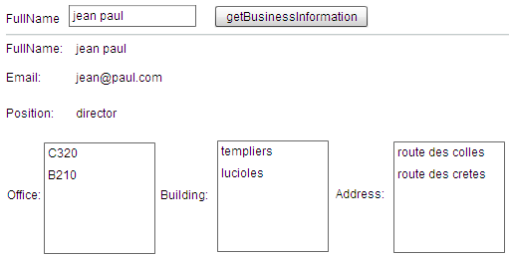


FIGURE 4.7 – IHM pour l'opération `getBusinessInfo`

Exemple 2 (IHM pour `getBusinessInfo`). $caui_{getBusinessInfo} \in app\_caui(BusinessApplication)$ correspond à la formalisation de cette IHM. Celle-ci correspond à :

$$
\begin{aligned}
caui\_id(caui_{getBusinessInfo}) &= getBusinessInfo_{ui} \\
caui\_events(caui_{getBusinessInfo}) &= \{event_1\} \\
event\_id(event_1) &= ui\_event_1 \\
event\_name(event_1) &= getBusinessInfo \\
event\_inputs(event_1) &= \{input_1\} \\
event\_outputs(event_1) &= \left\{ \begin{array}{l} output_1, output_2, output_3, \\ output_4, output_5, output_6 \end{array} \right\}
\end{aligned}
$$

X	$caui\_in\_id(X)$	$caui\_in\_name(X)$	$caui\_in\_type(X)$	$caui\_in\_cardinality(X)$
$input_1$	ui_input_1	$fullName$	$string$	1

X	$caui\_out\_id(X)$	$caui\_out\_name(X)$	$caui\_out\_type(X)$	$caui\_out\_cardinality(X)$
$output_1$	ui_output_1	$address$	$string$	n
$output_2$	ui_output_2	$building$	$string$	n
$output_3$	ui_output_3	$email$	$string$	1
$output_4$	ui_output_4	$fullName$	$string$	1
$output_5$	ui_output_5	$office$	$string$	n
$output_6$	ui_output_6	$position$	$string$	1

Il n'y a pas de valeur pour la sélection de l'entrée puisque celle-ci a une cardinalité égale à 1.

Liens d'interaction entre le composant fonctionnel et les composants d'IHM

Les liens d'interaction décrivent les échanges de données entre le composant fonctionnel et les composants d'IHM mais également l'émission d'événement de la part d'un des composants d'IHM pour déclencher l'appel d'une action du composant fonctionnel. Trois fonctions définissent ces liens : *DataLinkII*, *DataLinkOO* et *EventLink* fournit par la définition 4.

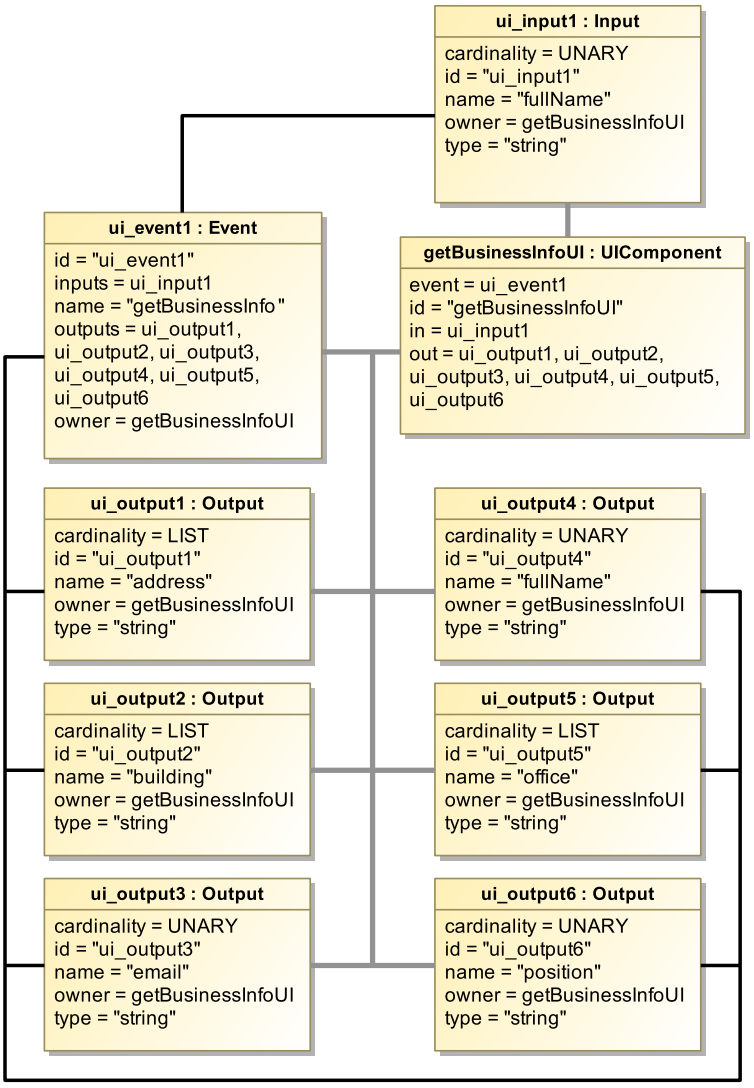


FIGURE 4.8 – Instantiation de l'IHM `getBusinessInfoUI` au sein du méta-modèle `AliasComponent`

Définition 4 (Liens d'interaction)**.** Les liens d'interaction sont définis par :

- $DataLinkII : CAUI \times CFC \rightarrow caui\_inputs(CAUI) \times cfc\_inputs(CFC)$
  $\forall caui \in app\_caui(appli), datalinkII(caui, cfc) = \{(i_1, i_2)\}$
- $DataLinkOO : CAUI \times CFC \rightarrow caui\_outputs(CAUI) \times cfc\_outputs(CFC)$
  $\forall caui \in app\_caui(appli), datalinkOO(caui, cfc) = \{(o_1, o_2)\}$
- $EventLink : CAUI \times CFC \rightarrow caui\_events(CAUI) \times cfc\_actions(CFC)$
  $\forall caui \in app\_caui(appli), eventlink(caui, cfc) = \{(ev, act)\}$

Ainsi, dans le cas de $dataLinkII$, l'élément i_1 est la représentation graphique de la donnée d'entrée i_2, dans le cas de $dataLinkOO$, la donnée o_2 est représenté par o_1 et pour finir dans $eventLink$, evt déclenche act.

Les contraintes liées aux composants fonctionnels et d'IHM ont été définies auparavant et se retrouvent au sein de la définition d'une application (cf. listing 4.1, 4.2 et 4.3). Les règles OCL suivantes permettent d'avoir un modèle conforme à la formalisation qui a été faite d'une application et contraignent les liens d'interaction :

– les liens qui relient deux entrées entre elles doivent se faire depuis un port d'entrée d'un composant d'IHM vers un port d'entrée du composant fonctionnel (cf. listing 4.4)

```
context Application inv dataLinkII :
self.dataLinkII->forAll(
    (from.oclIsTypeOf(Input) and to.oclIsTypeOf(Input)) and
    (from.oclAsType(Input).owner.oclIsTypeOf(UIComponent) and
     to.oclAsType(Input).owner.oclIsTypeOf(FCComponent)))
```

Listing 4.4 – Règle OCL qui décrit les liens entre deux entrées

– les liens qui relient deux sorties entre elles doivent se faire depuis un port de sortie du composant fonctionnel vers un port de sortie d'un composant d'IHM (cf. listing 4.5)

```
context Application inv dataLinkOO :
self.dataLinkOO->forAll(
    (from.oclIsTypeOf(Output) and to.oclIsTypeOf(Output)) and
    (from.oclAsType(Output).owner.oclIsTypeOf(FCComponent) and
     to.oclAsType(Output).owner.oclIsTypeOf(UIComponent)))
```

Listing 4.5 – Règle OCL qui décrit les liens entre deux sorties

– les liens qui décrivent le déclenchement de l'appel d'une opération par un élément de l'IHM relient un *EVENT* du composant d'IHM à une *ACTION* du composant fonctionnel (cf. listing 4.6)

```
context Application inv eventLink :
self.eventLink->forAll(
    (from.oclIsTypeOf(Event) and to.oclIsTypeOf(Action)) and
    (from.oclAsType(Event).owner.oclIsTypeOf(UIComponent) and
     to.oclAsType(Action).owner.oclIsTypeOf(FCComponent)))
```

Listing 4.6 – Règle OCL qui décrit les liens entre un *Event* et une *Action*

Exemple. On illustre la description d'une application ainsi que des liens d'interaction sur l'application *BusinessApplication* présentée en section 2.1.1. Cette application comporte une partie fonctionnelle et deux IHM (dont une des IHM a été illustrée auparavant (cf. figure 4.7) ainsi que les différents liens de données et d'événements qui existent entre les différentes parties. La partie fonctionnelle correspond à un *Web Service* et a été illustrée dans l'exemple 1. Elle possède deux opérations : `getBusinessInfo` et `getAddresses`. Les deux IHM permettent d'interagir avec chacune des deux opérations. La première IHM a été illustrée dans l'exemple 2 lors de la description du composant d'IHM. La seconde IHM permet d'interagir avec l'opération `getAddresses` et est présentée en figure 4.9 et est formalisée dans l'exemple 3.

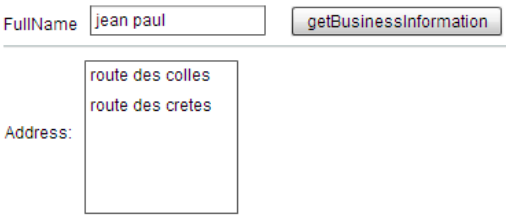


FIGURE 4.9 – IHM pour l'opération `getAddresses`

Exemple 3 (IHM pour `getAddresses`). $caui_{getAddresses} \in app\_caui(BusinessApplication)$ correspond à la formalisation de cette IHM. Celle-ci correspond à :

$$\begin{aligned} caui\_id(caui_{getAddresses}) &= getAddresses_{ui} \\ caui\_events(caui_{getAddresses}) &= \{event_1\} \\ event\_id(event_1) &= ui\_event_1 \\ event\_name(event_1) &= getAddresses \\ event\_inputs(event_1) &= \{input_1\} \\ event\_outputs(event_1) &= \{output_1\} \end{aligned}$$

X	$caui\_in\_id(X)$	$caui\_in\_name(X)$	$caui\_in\_type(X)$	$caui\_in\_cardinality(X)$
$input_1$	ui_input_1	$fullName$	$string$	1
X	$caui\_out\_id(X)$	$caui\_out\_name(X)$	$caui\_out\_type(X)$	$caui\_out\_cardinality(X)$
$output_1$	ui_output_1	$addresses$	$string$	n

De la même manière que précédemment, il n'y a pas de valeur pour la sélection de l'entrée puisque celle-ci a une cardinalité égale à 1.

Le résultat de la formalisation de l'application *BusinessApplication* est fourni par l'exemple 4 et sa représentation au sein du méta-modèle `AliasComponent` dans la figure 4.10.

Exemple 4 (*BusinessApplication*). $BusinessApplication$ correspond à la formalisation de cette application avec :

$$\begin{aligned} app\_id(BusinessApplication) &= BusinessInfo_{application} \\ app\_cfc(BusinessApplication) &= cfc_{BusinessService} \\ app\_caui(BusinessApplication) &= \{caui_{getBusinessInfo}, caui_{getAddresses}\} \\ dataLinkII(caui_{getBusinessInfo}, cfc_{BusinessService}) &= \{(input_1, input_1)\} \\ dataLinkOO(caui_{getBusinessInfo}, cfc_{BusinessService}) &= \left\{ \begin{array}{l} (output_1, output_1), \\ (output_2, output_2), \\ (output_3, output_3), \\ (output_4, output_4), \\ (output_5, output_5), \\ (output_6, output_6) \end{array} \right\} \\ eventLink(caui_{getBusinessInfo}, cfc_{BusinessService}) &= \{(event_1, action_1)\} \\ dataLinkII(caui_{getAddresses}, cfc_{BusinessService}) &= \{(input_1, input_2)\} \\ dataLinkOO(caui_{getAddresses}, cfc_{BusinessService}) &= \{(output_1, output_7)\} \\ eventLink(caui_{getAddresses}, cfc_{BusinessService}) &= \{(event_1, action_2)\} \end{aligned}$$

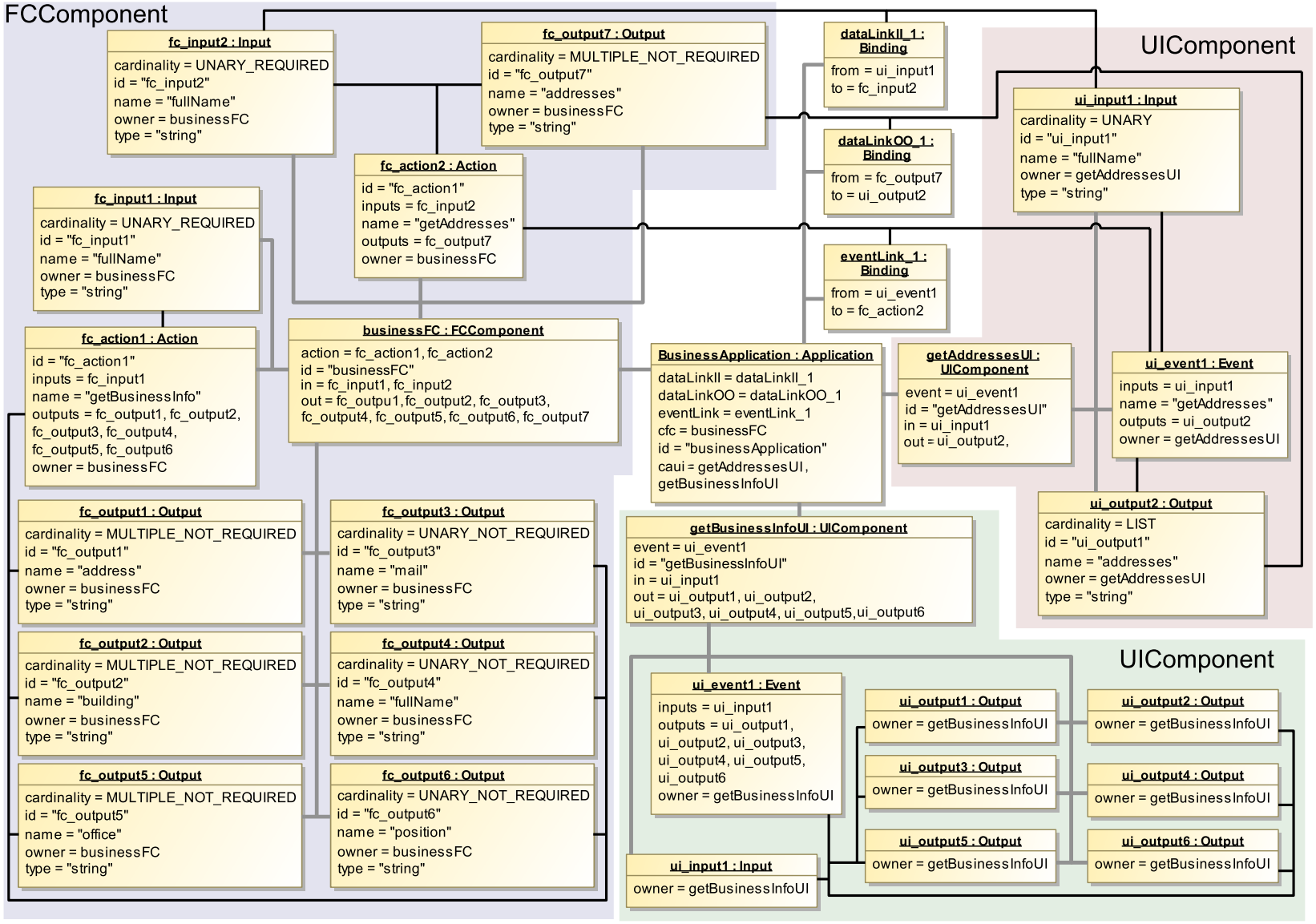


FIGURE 4.10 – Illustration d'un extrait de l'application *Business* avec `AliasComponent`

Propriétés de la formalisation. Ce qui suit présente les propriétés que l'on pose sur la formalisation concernant la description des applications.

Propriété 1. *Tous les ports (de données ou d'événements) des composants d'IHM d'une application doivent être utilisés car les composants ne conservent que les éléments d'IHM qui interagissent avec la partie fonctionnelle. Donc l'ensemble des ports (d'entrée, de sortie et d'événement) doit être relié au composant fonctionnel. Ceci est décrit par les trois propriétés suivantes :*

$$\forall caui \in app\_caui(appli), caui\_inputs(caui) = \left\{ i \;\middle|\; \begin{array}{l} \exists i1 \in cfc\_inputs(app\_cfc(appli)) \wedge \\ (i, i1) \in dataLinkII(caui, app\_cfc(appli)) \end{array} \right\} \quad (4.9)$$

$$\forall caui \in app\_caui(appli), caui\_outputs(caui) = \left\{ o \;\middle|\; \begin{array}{l} \exists o1 \in cfc\_outputs(app\_cfc(appli)) \wedge \\ (o, o1) \in dataLinkOO(caui, app\_cfc(appli)) \end{array} \right\} \quad (4.10)$$

$$\forall caui \in app\_caui(appli), caui\_events(caui) = \left\{ e \;\middle|\; \begin{array}{l} \exists a \in cfc\_actions(app\_cfc(appli)) \wedge \\ (e, a) \in eventLink(caui, app\_cfc(appli)) \end{array} \right\} \quad (4.11)$$

Propriété 2. *Tous les ports de données d'entrée et de sortie associés à une action utilisée doivent être reliés à des ports de données d'entrée et de sortie d'un même composant d'IHM. Ce qui est défini par :*

$$\forall (event, action) \in eventLink(caui, cfc) = \left\{ i \in action\_inputs(action) \;\middle|\; \begin{array}{l} \exists i1 \in caui\_input(caui) \wedge \\ (i1, i) \in dataLinkII(caui, cfc) \end{array} \right\} \quad \begin{array}{l} \forall caui \in app\_caui(appli), \\ \\ \end{array} \quad (4.12)$$

$$\forall (event, action) \in EventLink(caui, cfc) = \left\{ o \in outputs(action) \;\middle|\; \begin{array}{l} \exists o1 \in output(caui) \wedge \\ (i1, i) \in DataLinkOO(caui, cfc) \end{array} \right\} \quad \begin{array}{l} \forall caui \in caui(appli), \\ \\ \end{array} \quad (4.13)$$

Propriété 3. *Les liens de données doivent relier des éléments qu'il est possible de connecter entre eux, c'est-à-dire que l'arité des éléments de destination est supérieur ou égale à celle de la source. Dans le cas de liens entre une IHM et un composant fonctionnel, on obtient ensuite les propriétés suivantes avec $cfc = app\_cfc(appli)$:*

$$\forall caui \in app\_caui(appli), \forall (p1, p2) \in dataLinkII(caui, cfc),$$

$$\left\{ \begin{array}{l} caui\_in\_cardinality(p1) = 1 \Rightarrow cfc\_in\_cardinality(p2) = [0,1] \vee [1,1] \\ \left(\begin{array}{r} caui\_in\_cardinality(p1) = 1 \wedge \\ caui\_in\_selection(p1) = single \end{array} \right) \Rightarrow cfc\_in\_cardinality(p2) = [0,1] \vee [1,1] \\ \left(\begin{array}{r} caui\_in\_cardinality(p1) = n \wedge \\ caui\_in\_selection(p1) = multiple \end{array} \right) \Rightarrow cfc\_in\_cardinality(p2) = [0,n] \vee [1,n] \end{array} \right.$$

$$\forall caui \in app\_caui(appli), \forall (p1, p2) \in dataLinkOO(caui, cfc),$$

$$\left\{ \begin{array}{l} cfc\_out\_cardinality(p2) = [0,1] \vee [1,1] \Rightarrow caui\_out\_cardinality(p1) = 1 \vee n \\ cfc\_out\_cardinality(p2) = [0,n] \vee [1,n] \Rightarrow caui\_out\_cardinality(p1) = n \end{array} \right.$$

Les propriétés 1, 2 et 3 doivent être vérifiées lors des transformations d'abstraction d'une application.

4.2.2 La composition fonctionnelle

La composition fonctionnelle décrit l'usage qui est fait des composants fonctionnels composés comme on peut l'avoir dans les orchestrations de *Web Services* (cf. section 3.1.1) ou dans les assemblages de composants (cf. section 3.1.2). Ainsi, la composition fonctionnelle décrit les échanges de données qu'il y a entre les opérations ainsi que l'enchaînement des appels d'opérations. Ces deux informations forment le flot de données et le flot de contrôle. Le flot de données permet de décrire des fusions (entre plusieurs entrées ou entre plusieurs sorties), du partage (une entrée/-sortie utilisée par plusieurs entrées/sorties) ou encore de l'utilisation de résultat (entre une sortie et une entrée). Le flot de contrôle permet de décrire des exécutions en parallèle ou en séquence, ou bien encore des conditions. L'ensemble de ces informations doivent se retrouver au sein d'un

composant composite qui correspond à la composition fonctionnelle car ces informations sont utilisées durant le processus de composition des IHM. Le résultat obtenu suite à la réalisation de la composition fonctionnelle est un nouveau composant fonctionnel. Ceci se résume de la manière suivante (où $\otimes$ correspond aux flots de données et de contrôle, cette fonction est associative mais pas commutative) :

$$NF_1 \otimes NF_2 \otimes ... \otimes NF_n \Rightarrow NF$$

Définitions et propriétés de la composition fonctionnelle

La composition fonctionnelle correspond à un composant composite fonctionnel dont les sous-composants fonctionnels correspondent aux services ou composants que l'on compose. Elle permet d'identifier à la fois le flot de données échangées entre les composants fonctionnels ainsi qu'entre les composants fonctionnels et le composant composite fonctionnel. Elle permet également d'identifier le flot de contrôle. Ce flot de contrôle décrit l'enchaînement des différentes opérations à appeler au cours de l'exécution de la composition fonctionnelle. Dans notre cas, le résultat de la composition fonctionnelle est un nouveau composant fonctionnel qui possède les mêmes caractéristiques qu'en définition 2. Ce résultat est le produit de l'utilisation des liens de données et d'actions qui existent entre le résultat de la composition fonctionnelle et les composants fonctionnels impliqués dans la composition. Ils vont ainsi permettre de déterminer les entrées, sorties et actions que possédera ce nouveau composant. Le composant résultant de la composition sera nommé indifféremment : composant résultant, cfc_{res} ou bien encore composant composite (puisque de la même manière que dans les approches à composants, il agrège un ensemble de composant en son sein).

Formalisation. La définition 5 correspond à la formalisation de la composition fonctionnelle qui crée un composant composite fonctionnel et l'assemblage avec et entre les sous-composants fonctionnels.

Définition 5 (ComposeFC)**.** La composition fonctionnelle est définie par la fonction de composition *ComposeFC* qui à partir d'un ensemble d'applications retourne un nouveau composant composite fonctionnel.

$$ComposeFC : APPLI^{+} \rightarrow CFC \tag{4.14}$$

$$composeFC(\cup_{i=1}^{n} appli_i) = cfc_{res} \tag{4.15}$$

- $APPLI^{+}$: l'ensemble des applications impliquées dans la composition fonctionnelle et plus spécifiquement les $cfc(appli_i)$ correspondant au composant fonctionnel de chaque $appli_i$
- cfc_{res} : le composant composite fonctionnel obtenu par la fonction de composition *ComposeFC*. Ce composant composite fonctionnel est donné par la fonction : $composeFC(\cup_{i=1}^{n} appli_i)$

Un effet de bord de cette fonction de composition est la création de liens entre les sous-composants fonctionnels qui décrivent les échanges de données entre une sortie et une entrée et le flot de contrôle entre deux actions (cf. définition 6)

Définition 6 (Liens entre sous-composants fonctionnels)**.** *SCALink* décrit les liens entre deux actions des sous-composants fonctionnels et *SCDLinkOI* décrit les liens qu'il y a entre une sortie et une entrée des sous-composants fonctionnels.

- *SCALink* correspond au flot de contrôle et décrit les liens d'action entre deux actions ($action_1$ and $action_2$) des sous-composants fonctionnels

$$SCALink : CFC \times CFC \rightarrow ACTION \times ACTION \tag{4.16}$$

$$\exists cfc_1, \exists cfc_2 \in \cup_{i=1}^{n} app\_cfc(appli_i),$$

$$scalink(cfc_1, cfc_2) = \left\{ (action_1, action_2) \;\middle|\; \begin{array}{l} action_1 \in cfc\_actions(cfc_1) \wedge \\ action_2 \in cfc\_actions(cfc_2) \end{array} \right\} \tag{4.17}$$

– *SCDLinkOI* décrit que le paramètre de sortie o_1 du composant fonctionnel cfc_1 est un paramètre d'entrée i_1 d'un autre composant fonctionnel cfc_2.

$$SCDLinkOI : CFC \times CFC \rightarrow OUT \times IN \tag{4.18}$$

$$\exists cfc_1, \exists cfc_2 \in \cup_{i=1}^{n} app\_cfc(appli_i),$$

$$scdlinkOI(cfc_1, cfc_2) = \left\{ (o_1, i_1) \;\middle|\; \begin{array}{l} o_1 \in cfc\_outputs(cfc_1) \wedge \\ i_1 \in cfc\_inputs(cfc_2) \end{array} \right\} \tag{4.19}$$

On définit également des liens qui relient le composant composite fonctionnel cfc_{res} à ses sous-composant fonctionnels (cf. définition 7). Ces liens sont présents dans la définition d'un composant fonctionnel (*FCComponent*) et sont décrits par trois liens distincts : *TOSCALink*, *TOSCDLinkII* and *TOSCDLinkOO*.

Définition 7 (Liens internes au composant fonctionnel)**.** Les fonctions suivantes définissent les liens qui relient les ports du composant composite fonctionnel cfc_{res} à ses sous-composants fonctionnels. Ici, cfc_{res} correspond au composant composite fonctionnel qui est le résultat de la fonction de composition *ComposeFC* (d'après la définition 5 où $cfc_{res} = composeFC(\cup_{i=1}^{n} appli_i)$).

$$TOSCALink : CFC \times CFC \rightarrow ACTION \times ACTION \tag{4.20}$$

$$\exists cfc_2 \in \cup_{i=1}^{n} app\_cfc(appli_i),$$

$$toscalink(cfc_{res}, cfc_2) = \left\{ (action_1, action_2) \;\middle|\; \begin{array}{l} action_1 \in cfc\_actions(cfc_{res}) \wedge \\ action_2 \in cfc\_actions(cfc_2) \end{array} \right\} \tag{4.21}$$

$$TOSCDLinkII : CFC \times CFC \rightarrow INPUT \times INPUT \tag{4.22}$$

$$\exists cfc_2 \in \cup_{i=1}^{n} app\_cfc(appli_i),$$

$$toscdlinkII(cfc_{res}, cfc_2) = \left\{ (input_1, input_2) \;\middle|\; \begin{array}{l} input_1 \in cfc\_inputs(cfc_{res}) \wedge \\ input_2 \in cfc\_inputs(cfc_2) \end{array} \right\} \tag{4.23}$$

$$TOSCDLinkOO : CFC \times CFC \rightarrow OUTPUT \times OUTPUT \tag{4.24}$$

$$\exists cfc_1 \in \cup_{i=1}^{n} app\_cfc(appli_i),$$

$$toscdlinkOO(cfc_1, cfc_{res}) = \left\{ (output_1, output_2) \;\middle|\; \begin{array}{l} output_1 \in cfc\_outputs(cfc_1) \wedge \\ output_2 \in cfc\_outputs(cfc_{res}) \end{array} \right\} \tag{4.25}$$

Dans le cadre de ce travail, trois opérateurs présents dans les orchestrations de services [KMW03] ont été utilisés et permettent de décrire le flot de contrôle. Ces trois opérateurs sont les suivants :

– La séquence : l'exécution des opérations se fait l'une à la suite de l'autre (cf. propriété 4),
– Le parallélisme : l'exécution de n opérations est faite en même temps (cf. propriété 5),
– La condition : l'exécution d'une opération est conditionnée par le résultat d'une opération précédente (cf. définition 10).

Ces opérateurs qui sont un sous-ensemble des opérateurs disponibles, permettent d'en décrire d'autres. Ainsi, il est par exemple possible de définir une boucle en utilisant une condition qui est relié à l'élément qui vient d'être exécuté.

Pour définir la séquence et le parallélisme, on introduit une fonction intermédiaire qui pour une action donnée retourne l'ensemble des actions qui la succède (cf. définition 8).

Définition 8 (Successor)**.** La fonction *Successor* retourne l'ensemble des actions qui succèdent une action ($action_1$). On définit $cfc_{res} = composeFC(\cup_{i=1}^{n} appli_i)$Elle est définie par :

$$Successor : ACTION \rightarrow ACTION^* \tag{4.26}$$

$$\forall cfc_1 \in \cup_{i=1}^{n} app\_cfc(appli_i), \forall action_1 \in cfc\_actions(cfc_1),$$

$$successor(action_1) = \left\{ action \;\middle|\; \begin{array}{l} \exists cfc \in \cup_{i=1}^{n} app\_cfc(appli_i) \wedge \\ (action_1, action) \in scalink(cfc_1, cfc) \end{array} \right\} \tag{4.27}$$

$$\forall action_1 \in cfc_{res},$$

$$successor(action_1) = \left\{ action \;\middle|\; \begin{array}{l} \exists cfc \in \cup_{i=1}^{n} app\_cfc(appli_i) \wedge \\ (action_1, action) \in toscalink(cfc_{res}, cfc) \end{array} \right\} \tag{4.28}$$

On introduit également la fonction *ConditionalSuccessor* (cf. définition 9) qui est permet d'obtenir deux ensembles de successeurs en fonction d'une condition donnée. On enrichit ainsi la fonction *Successor*.

Définition 9 (ConditionalSuccessor)**.** La fonction *ConditionalSuccessor* retourne ainsi deux ensembles d'actions en fonction de la valeur de *Bool* qui peut être soit *true* soit $false$.

$$ConditionalSuccessor : Action \times Bool \rightarrow \begin{cases} Action_A^* \text{ si } Bool = true \\ Action_B^* \text{ si } Bool = false \end{cases} \tag{4.29}$$

L'ensemble des successeurs $Action_A$ et $Action_B$ possèdent les mêmes propriétés que celles définies pour la fonction *Successor* (cf. définition 8).

Propriété 4 (Sequence)**.** *D'après la définition 8 et en réutilisant les fonctions 4.27 et 4.28. Il existe une séquence entre $action_1$ et $action_2$ appartenant à l'ensemble des actions présentes lorsqu'il n'y qu'un seul et unique successeur à $action_1$ qui est $action_2$.*

$$\exists action_1, \exists action_2 \in cfc\_actions(app\_cfc(\cup_{i=1}^{n} appli_i) \cup composeFC(\cup_{i=1}^{n} appli_i))) \mid$$
$$action_2 \in successor(action_1) \wedge card(successor(action_1)) = 1$$

Propriété 5 (Parallelism)**.** *D'après la définition 8 et en réutilisant les fonctions 4.27 et 4.28. Un ensemble d'actions : $sub\_action = \{action_a, \ldots, action_z\}$ sont en parallèle lorsqu'elles ont un unique prédécesseur $action_{init}$.*

$$\exists action_{init}, \exists sub\_action \in cfc\_actions(app\_cfc(\cup_{i=1}^{n} appli_i) \cup composeFC(\cup_{i=1}^{n} appli_i))) \mid$$
$$sub\_action = successor(action_{init}) \wedge card(successor(action_{init}) \geq 2$$

Définition 10 (Condition)**.** On définit ainsi la condition qui pour une action donnée et une fonction booléenne de test appliquée à une valeur (*Val*) retourne deux ensembles de successeurs calculé par la fonction *ConditionalSuccessor*.

$$Condition : Action \times test(Val) = ConditionalSuccessor(Action, test(Val)) \tag{4.30}$$
$$condition(action_1, test(val)) = conditionSuccessor(action_1, test(val)) \tag{4.31}$$

Avec :

$$action_1 \in cfc\_actions(\cup_{i=1}^{n} app\_cfc(appli_i) \cup composeFC(\cup_{i=1}^{n} appli_i))$$
$$val \in action\_inputs(action_1) \oplus val \in action\_outputs(action_1)$$

Dans la formalisation, on a distingué les différents types de liens (entre sous-composants fonctionnels ou entre le composant composite fonctionnel et ses sous-composants fonctionnels). Ceci n'est pas le cas au sein de la méta-modélisation où il n'y a pas de distinction de faite. Ainsi, on a l'équivalence suivante : *SCALink* et *TOSCALink* correspondent à des *actionLink*, *SCDLinkOI* correspond à *dataLinkOI*, *TOSCADLinkII* correspond à *dataLinkII* et pour finir *TOSCADlinkOO* correspond à *dataLinkOO*. On retrouve dans le méta-modèle le fait que le composant composite fonctionnel (*FCComposition*) possèdent les mêmes caractéristiques qu'un composant fonctionnel (*FCComponent*) et qu'il en agrège plusieurs en son sein (*cfc*).

Méta-modélisation. La figure 4.11 représente la composition fonctionnelle au sein du méta-modèle `AliasComponent`. La composition fonctionnelle (*FCComposition*) hérite du composant fonctionnel. Elle possède en plus un ensemble de sous-composants fonctionnels (*subcomponents*) et des liens de données et d'actions. Les liens (*Bindings*) sont contraints de la même manière que dans l'application afin qu'ils correspondent aux liens décrits dans la définition 5. Ces contraintes sont appliquées au composant fonctionnel de telle sorte que les quatre listes de liens qu'il possède

soient conformes à la formalisation qui en a été faite et donc que la liste *dataLinkII* relie les entrées du composant composite à des entrées de ses sous-composants, que la liste *dataLinkOO* relie des sorties de sous-composants aux sorties du composant composite, que la liste *dataLinkOI* relie des sorties des sous-composants à des entrées d'un ou plusieurs autres sous-composants et enfin que la liste *actionLink* relie des actions du composant composite à des actions de ses sous-composants ou des actions des sous-composants entre eux. Les contraintes associées aux liens permettent également d'exprimer les liaisons vers une condition. Ces liaisons décrivent la réception de la valeur à tester ainsi que les exécutions des opérations qui suivent. Ce qui permet de pouvoir sélectionner, en fonction d'un test, les déclenchements d'appel d'opérations suivants.

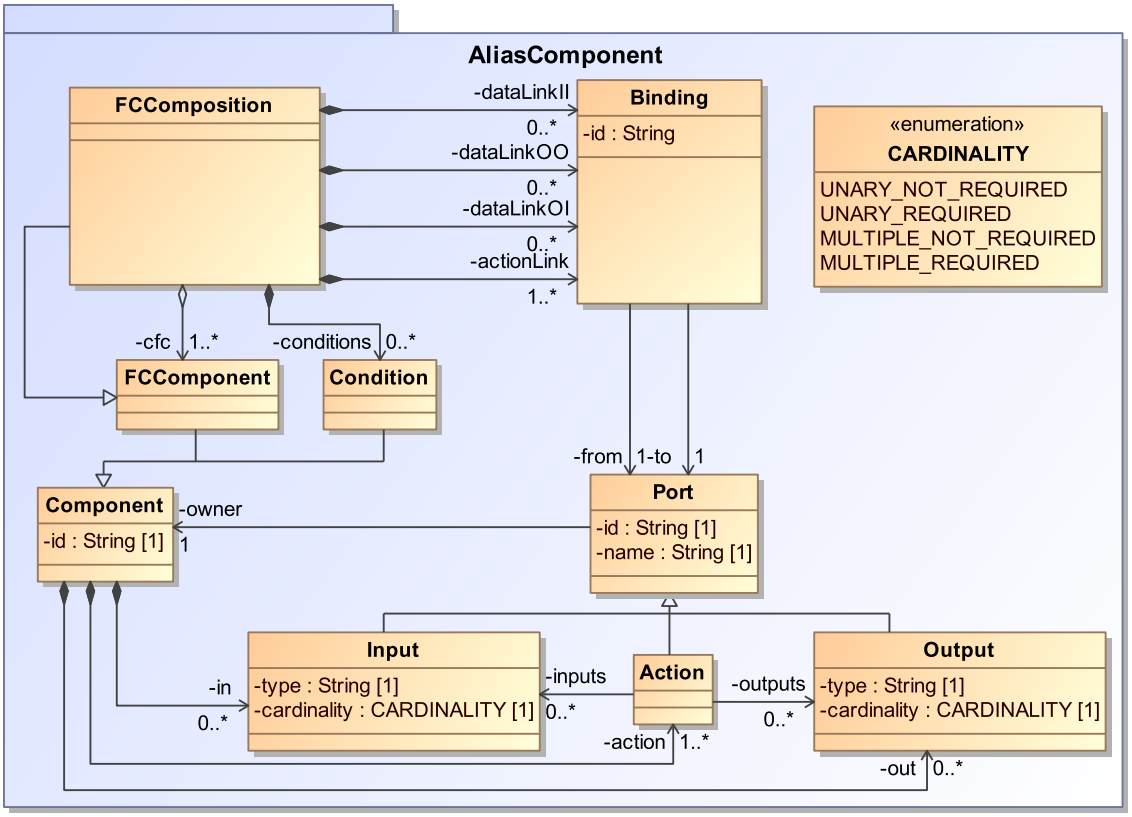


FIGURE 4.11 – Représentation de la composition fonctionnelle au sein d'`AliasComponent`

Les règles OCL qui permettent d'avoir un modèle conforme à la formalisation qui été faite d'une composition fonctionnelle sont les suivantes :

– les liens qui relient deux entrées entre elles doivent se faire depuis un port d'entrée du composant composite vers un port d'entrée d'un de ses sous-composants ou pour fournir une donnée à tester à une condition (cf. listing 4.7).

```
context FCComposition inv dataLinkII :
self.dataLinkII->forAll(
    (from.oclIsTypeOf(Input) and to.oclIsTypeOf(Input)) and
    ((from.oclAsType(Input).owner.oclIsTypeOf(FCComposition) and
      to.oclAsType(Input).owner.oclIsTypeOf(FCComponent)) xor
     (from.oclAsType(Input).owner.oclIsTypeOf(FCComposition) and
      to.oclAsType(Input).owner.oclIsTypeOf(Condition))))
```

Listing 4.7 – Règle OCL qui décrit les liens entre deux entrées

– les liens qui relient deux sorties entre elles doivent se faire entre une sortie d'un sous-composant avec une sortie du composant composite (cf. listing 4.8).

```
context FCCcomposition inv dataLinkOO :
self.dataLinkOO->forAll(
    (from.oclIsTypeOf(Output) and to.oclIsTypeOf(Output)) and
    (from.oclAsType(Output).owner.oclIsTypeOf(FCComponent) and
     to.oclAsType(Output).owner.oclIsTypeOf(FCComposition)))
```

Listing 4.8 – Règle OCL qui décrit les liens entre deux sorties

– les liens qui relient une sortie et une entrée entre elles doivent se faire entre une sortie d'un sous-composant et une entrée d'un sous-composant ou pour fournir une donnée à une condition (cf. listing 4.9).

```
context FCComposition inv dataLinkOI :
self.dataLinkII->forAll(
    (from.oclIsTypeOf(Output) and to.oclIsTypeOf(Input)) and
    ((from.oclAsType(Output).owner.oclIsTypeOf(FCComponent) and
      to.oclAsType(Input).owner.oclIsTypeOf(FCComponent)) xor
     (from.oclAsType(Output).owner.oclIsTypeOf(FCComponent) and
      to.oclAsType(Input).owner.oclIsTypeOf(Condition))))
```

Listing 4.9 – Règle OCL qui décrit les liens entre une sortie et une entrée

– les liens qui décrivent l'appel des opérations doivent se faire soit entre une *ACTION* du composant composite vers une *ACTION* d'un de ses sous-composants, soit entre deux *ACTION*s de sous-composants, soit depuis une *ACTION* du composant composite ou d'un sous-composant vers une condition, soit d'une condition vers des *ACTION*s de sous-composants (cf. listing 4.10).

```
context FCComposition inv actionLink :
self.actionLink->forAll(
    (from.oclIsTypeOf(Action) and to.oclIsTypeOf(Action)) and
    ((from.oclAsType(Action).owner.oclIsTypeOf(FCComposition) and
      to.oclAsType(Action).owner.oclIsTypeOf(FCComponent)) xor
     (from.oclAsType(Action).owner.oclIsTypeOf(FCComponent) and
      to.oclAsType(Action).owner.oclIsTypeOf(FCComponent)) xor
     (from.oclAsType(Action).owner.oclIsTypeOf(FCComposition) and
      to.oclAsType(Action).owner.oclIsTypeOf(Condition)) xor
     (from.oclAsType(Action).owner.oclIsTypeOf(FCComponent) and
      to.oclAsType(Action).owner.oclIsTypeOf(Condition)) xor
     (from.oclAsType(Action).owner.oclIsTypeOf(Condition) and
      to.oclAsType(Action).owner.oclIsTypeOf(FCComponent))))
```

Listing 4.10 – Règle OCL qui décrit les liens entre deux actions

Exemple. L'abstraction de la composition fonctionnelle est illustrée sur la troisième composition présentée en section 2.3.3. Cette composition a pour objectif d'obtenir l'ensemble des adresses d'une personne (professionnelle et personnelle) à partir du nom de la personne ainsi que de son numéro de Carte Vitale. Cette composition s'appuie sur le service *BusinessService* et sur le service *SocialInsuranceService*.

La figure 4.12 rappelle le *workflow* mis en place. Les deux opérations *getAddresses* du service *BusinessService* et *getByCard* du service *SocialInsuranceService* sont exécutées en parallèle.

La composition fonctionnelle possède deux entrées (cf. figure 4.13) : *CardID* et *FullName*. L'entrée *CardID* est reliée à l'entrée de l'opération *getByCard* et *FullName* à l'entrée de l'opération *getAddresses*.

La composition fonctionnelle possède une sortie *Addresses*. La valeur de la sortie est composée des valeurs de retours des deux opérations. Ainsi, on a une fusion des sorties. La liste fournie par l'opération *getAddresses* est augmentée de l'adresse présente dans les valeurs de retour de l'opération *getByCard* (cf. figure 4.14).

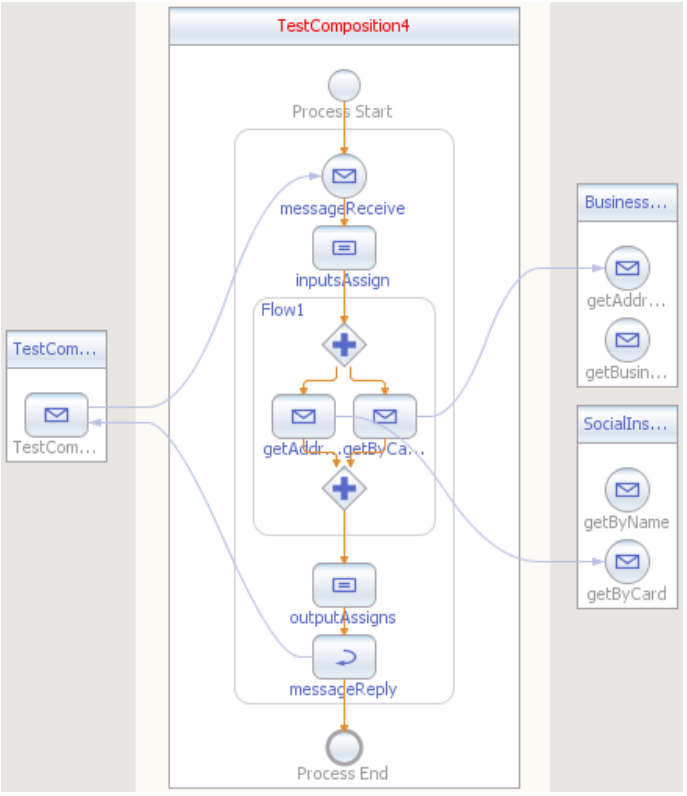


FIGURE 4.12 – Exécution en parallèle des deux opérations *getAddresses* et *getByCard*

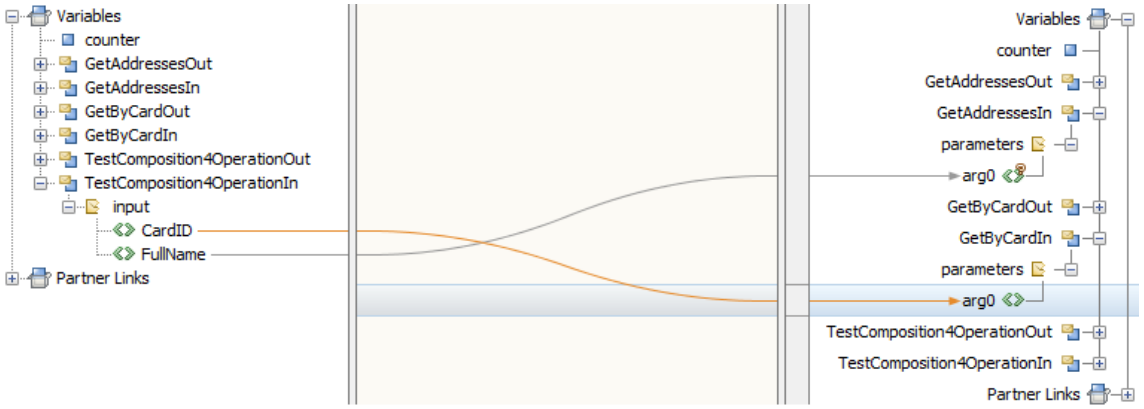


FIGURE 4.13 – Affectation des paramètres d'entrée

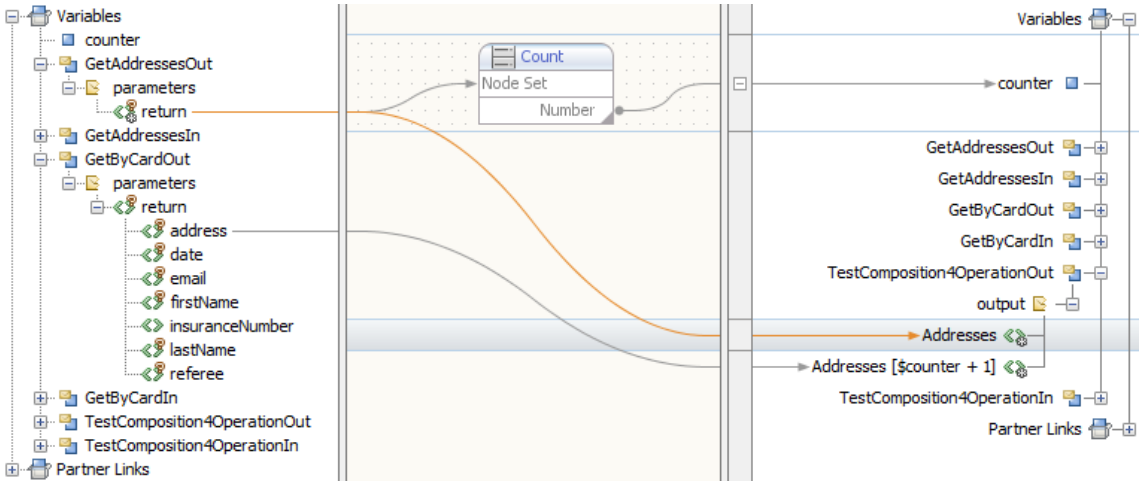


FIGURE 4.14 – Fusion des paramètres de sorties des deux opérations

Exemple 5 (Composant fonctionnel correspondant à la $3^{ème}$ composition fonctionnelle)**.** Le résultat de la composition est un nouveau composant fonctionnel $cfc_{Composition_3}$ qui est obtenu par la fonction *ComposeFC*. Pour cette composition on a donc :

$$cfc_{Composition_3} = composeFC(BusinessApplication, SocialInsuranceApplication)$$

Le composant $cfc_{Composition_3}$ possède les caractéristiques suivantes :

$$\begin{aligned} cfc\_id(cfc_{Composition_3}) &= composition_3 \\ cfc\_actions(cfc_{Composition_3}) &= caction_1 \\ action\_id(caction_1) &= cfc\_action_1 \\ action\_name(caction_1) &= getAddresses \\ action\_inputs(cation_1) &= \{cinput_1, cinput_2\} \\ action\_outputs(cation_1) &= \{coutput_2\} \end{aligned}$$

X	$cfc\_in\_id(X)$	$cfc\_in\_name(X)$	$cfc\_in\_type(X)$	$cfc\_in\_cardinality(X)$
$cinput_1$	cfc_input_1	$cardID$	$long$	$[1,1]$
$cinput_2$	cfc_input_2	$fullName$	$string$	$[1,1]$
X	$cfc\_out\_id(X)$	$cfc\_out\_name(X)$	$cfc\_out\_type(X)$	$cfc\_out\_cardinality(X)$
$coutput_1$	cfc_output_1	$addresses$	$string$	$[0,n]$

Par effet de bords, on obtient l'ensemble des liens entre le composant composite fonctionnel et ses sous-composants fonctionnels

$$\begin{aligned} \begin{pmatrix} app\_cfc(BusinessApplication) \cup \\ app\_cfc(SocialInsuranceApplication) \end{pmatrix} &= \{business_{fc}, socialInsurance_{fc}\} \\ toscdlinkII(cfc_{Composition_3}, socialInsurance_{fc}) &= \{(cinput_1, input_1)\} \\ toscdlinkII(cfc_{Composition_3}, business_{fc}) &= \{(cinput_2, input_2)\} \\ toscdlinkOO(cfc_{Composition_3}, socialInsurance_{fc}) &= \{(output_1, coutput_1)\} \\ toscdlinkOO(cfc_{Composition_3}, business_{fc}) &= \{(output_7, coutput_1)\} \end{aligned}$$

$$toscalink(cfc_{Composition_3}, socialInsurance_{fc}) = \{(caction_1, action_1)\}$$
$$toscalink(cfc_{Composition_3}, business_{fc}) = \{(caction_1, action_2)\}$$

où
- $input_1$ de $socialInsurance_{fc}$ correspond au numéro de carte de sécurité sociale de l'opération *getByCard*
- $input_2$ de $business_{fc}$ correspond au nom complet de l'opération *getAddresses*
- $output_1$ de $socialInsurance_{fc}$ correspond à l'adresse personnelle retournée par l'opération *getByCard*
- $output_7$ de $business_{fc}$ correspond à l'ensemble des adresses professionnelles retournées par l'opération *getAddresses*
- $action_1$ de $socialInsurance_{fc}$ correspond à l'opération *getByCard*
- $ation2_2$ de $business_{fc}$ correspond à l'opération *getAddresses*

Dans cette composition, les deux actions sont exécutées en parallèle. L'exemple 6 illustre le propriété de parallélisme (cf. propriété 5).

Exemple 6 (Parallélisme entre les deux actions de la composition fonctionnelle)**.** On montre le parallélisme entre les deux actions impliquées dans la composition fonctionnelle. Ainsi on a :

$$action_1 \in cfc\_actions(socialInsurance_{fc})$$
$$action_2 \in cfc\_actions(business_{fc})$$
$$\{action_1, action_2\} = successor(caction_1)$$
$$card(successor(caction_1)) = 2 \geq 2$$

Propriété de la formalisation. Ce composant fonctionnel abstrait cfc_{res}, produit par la fonction de composition *ComposeFC*, possède la propriété 6.

Propriété 6 (Intégrité de la composition fonctionnelle)**.** *Les liens de données doivent relier des éléments qu'il est possible de connecter entre eux, c'est-à-dire que la cardinalité des éléments est la même ou bien la destination possède une cardinalité plus grande que la source. On obtient ensuite les propriétés suivantes (avec* $cfc_{res} = composeFC(\cup_{i=1}^{n} appli_i)$*) :*

$$\forall cfc \in \cup_{i=1}^{n} app\_cfc(appli_i), \forall (p1, p2) \in toscdlinkII(cfc_{res}, cfc),$$
$$\begin{cases} cfc\_in\_cardinality(p1) = [0,1] \vee [1,1] \Rightarrow cfc\_in\_cardinality(p2) = [0,1] \vee [1,1] \vee [0,n] \vee [1,n] \\ cfc\_in\_cardinality(p1) = [0,n] \vee [1,n] \Rightarrow cfc\_in\_cardinality(p2) = [0,n] \vee [1,n] \end{cases}$$
$$\forall cfc \in \cup_{i=1}^{n} app\_cfc(appli_i), \forall (p1, p2) \in toscdlinkOO(cfc, cfc_{res}),$$
$$\begin{cases} cfc\_out\_cardinality(p2) = [0,1] \vee [1,1] \Rightarrow cfc\_out\_ccardinality(p1) = [0,1] \vee [1,1] \\ cfc\_out\_ccardinality(p2) = [0,n] \vee [1,n] \Rightarrow cfc\_out\_ccardinality(p1) = [0,1] \vee [1,1] \vee [0,n] \vee [1,n] \end{cases}$$
$$\forall cfc_1, cfc_2 \in \cup_{i=1}^{n} app\_cfc(appli_i), \forall (p1, p2) \in scdlinkOI(cfc_1, cfc_2),$$
$$\begin{cases} cfc\_out\_ccardinality(p1) = [0,1] \vee [1,1] \Rightarrow cfc\_in\_cardinality(p2) = [0,1] \vee [1,1] \vee [0,n] \vee [1,n] \\ cfc\_out\_ccardinality(p1) = [0,n] \vee [1,n] \Rightarrow cfc\_in\_cardinality(p2) = [0,n] \vee [1,n] \end{cases}$$

L'ensemble de ces propriétés doit être vérifié par la description de la composition fonctionnelle obtenue par abstraction ou directement décrite au sein d'`AliasComponent`.

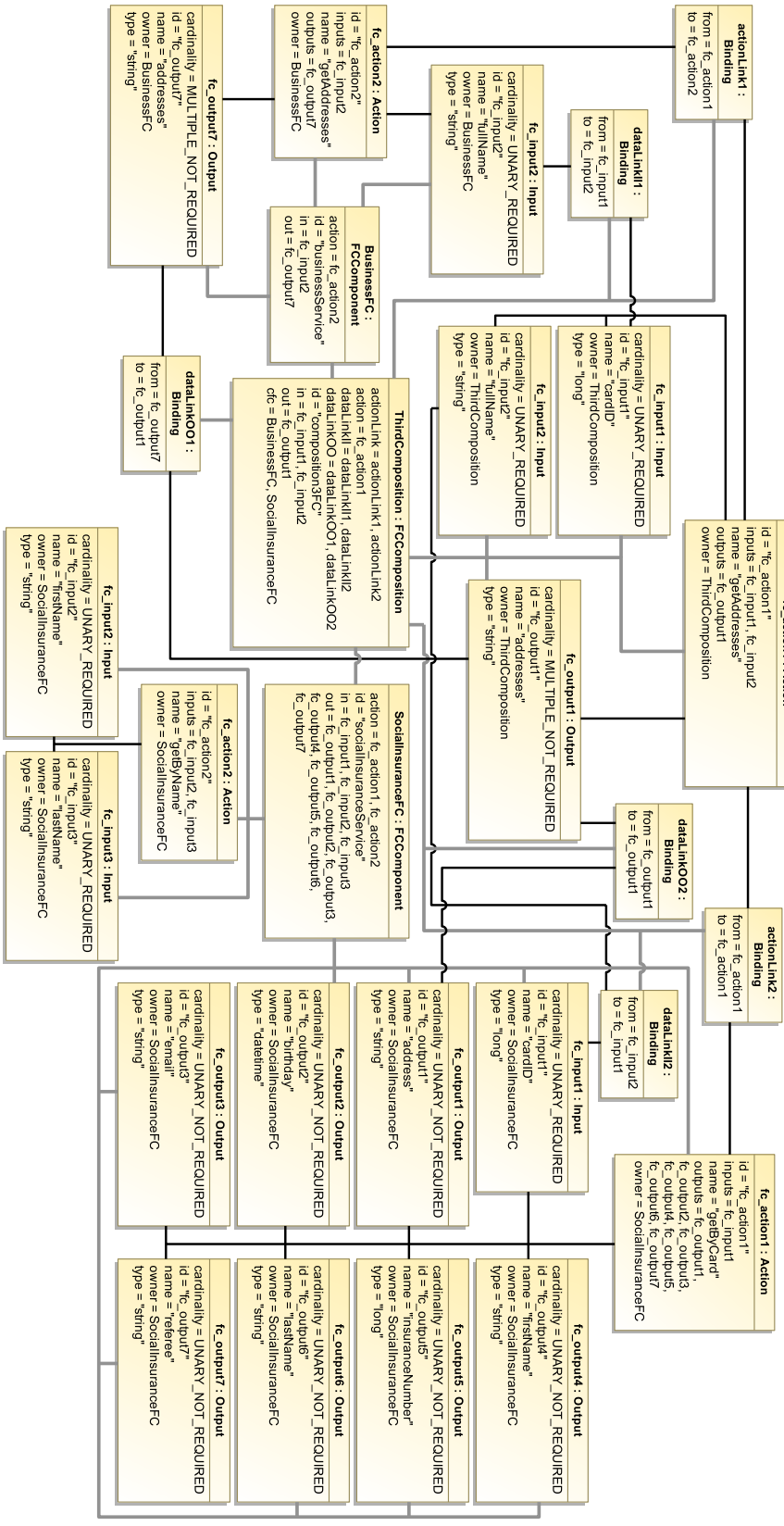


FIGURE 4.15 – Instantiation de la composition fonctionnelle au sein d'`AliasComponent`

4.2.3 La composition d'IHM

Basé sur la description abstraite des applications et de la composition fonctionnelle, il est maintenant possible de composer les IHM existantes. La fonction pour composer les IHM (cf. définition 11) requiert en entrée : une composition fonctionnelle et un ensemble d'applications abstraites. Cette fonction s'effectue en quatre étapes :

1. Identification des ports d'IHM abstraites. Pour chaque port du composant composite fonctionnel, on détermine les ports (d'entrée, sortie et d'événement) des composants d'IHM associés et on les relie ensemble.
2. Détection des conflits. Pour chaque point de fusion, c'est-à-dire lorsqu'une entrée du composant composite fonctionnel est partagée entre plusieurs entrées de ses sous-composants fonctionnels ou bien lorsque plusieurs sorties des sous-composants fonctionnels sont fusionnées pour en obtenir qu'une seule, on vérifie si les ports d'IHM sont identiques ou pas.
3. Résolution des conflits. Le développeur choisit manuellement une des possibilités de résolution de conflits : (i) sélectionner un des ports d'IHM parmi les n différents ports possibles, (ii) conserver l'ensemble des ports d'IHM ou (iii) créer un nouveau port d'IHM (en réutilisant des informations parmi les ports en conflit ou bien en fournissant lui-même les informations nécessaires).
4. Finalisation de la composition et création des composants d'IHM et des liens entre les composants d'IHM résultants et le composant composite fonctionnel.

Formalisation. La fonction de composition des IHM existantes, *ComposeUI*, est fournie par la définition 11. Elle permet d'obtenir une nouvelle application à partir d'un ensemble d'application et d'une composition fonctionnelle.

Définition 11 (ComposeUI)**.** La composition d'IHM est donnée par la fonction suivante :

$$ComposeUI : APPLI^{+} \times ComposeFC \rightarrow APPLI \tag{4.32}$$

$$composeUI \left(\begin{array}{l} \cup_{i=1}^{n} appli_i, \\ composeFC(\cup_{i=1}^{n} appli_i) \end{array} \right) = appli_{composed} \tag{4.33}$$

avec en entrée :

- $composeFC(\cup_{i=1}^{n} appli_i)$: la fonction de composition qui déclenche la composition d'IHM (cf. définition 5),
- $\cup_{i=1}^{n} appli_i$: l'ensemble des applications impliquées dans la composition fonctionnelle (cf. définition 1). Ces applications sont nécessaires pour déduire les ports d'IHM à conserver.

Méta-modélisation. La figure 4.16 reprend l'ensemble des éléments qui composent le méta-modèle `AliasComponent`. Ainsi on peut voir qu'une application peut posséder comme composant fonctionnel une composition fonctionnelle (puisque *FCComposition* hérite de *FCComponent*). Les contraintes ne sont pas rappelées mais elles s'appuient sur l'ensemble des contraintes définies auparavant (cf. listing 4.1, 4.2, 4.3, 4.4, 4.5 et 4.6).

On ajoute une nouvelle contrainte OCL afin de décrire les liens internes. Ceci est fourni par le listing 4.11.

```
context Application inv consistencyLink :
self.uiInternalLink->forAll(
  (from.owner.oclIsTypeOf(UIComponent) and to.owner.oclIsTypeOf(UIComponent)) and
  ((from.oclIsTypeOf(Input) and to.oclIsTypeOf(Input)) xor
   (from.oclIsTypeOf(Output) and to.oclIsTypeOf(Output)) xor
   )from.oclIsTypeOf(Event) and to.oclIsTypeOf(Event))))
```

Listing 4.11 – Règle OCL qui décrit les liens internes à un composant d'IHM

Ce qui suit présente l'ensemble des étapes de composition au travers de la formalisation qui est au cœur de ce processus.

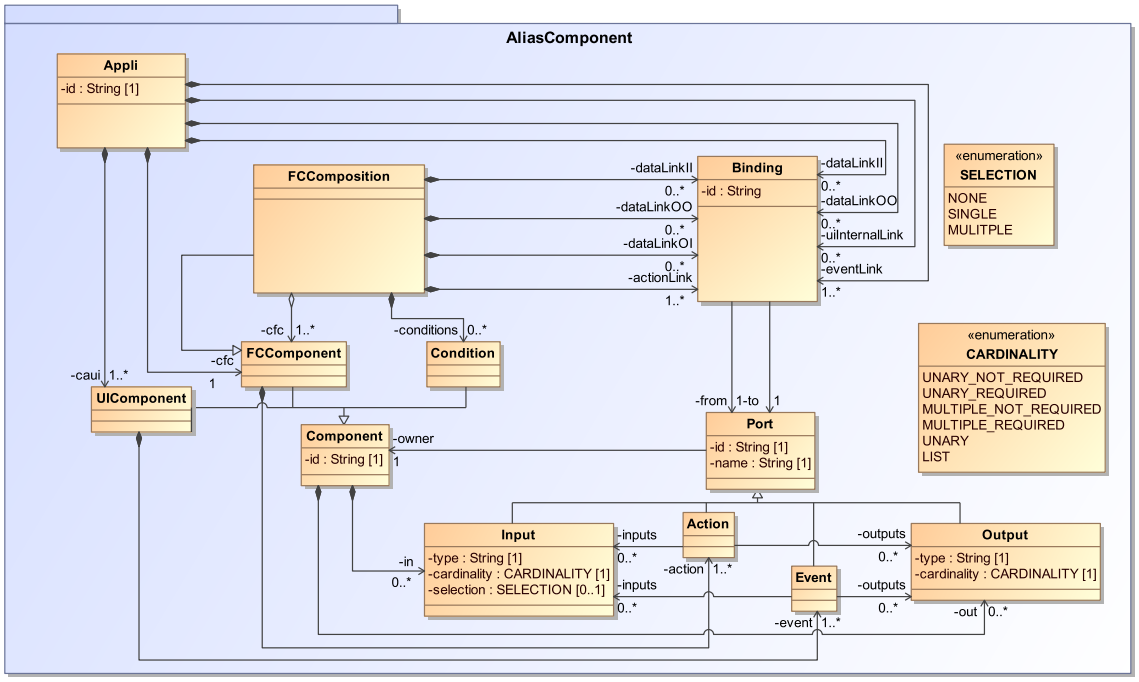


FIGURE 4.16 – Méta-modèle `AliasComponent`dans son intégralité

Première étape : déterminer les ports d'IHM à conserver

Dans cette étape, l'objectif est de déterminer les ports des composants d'IHM qui doivent être présents dans les composants d'IHM résultants à la fin du processus de composition (cf. définition 7). Ces ports sont déterminés en utilisant la composition fonctionnelle cfc_{res} (résultat de la fonction $composeFC(\cup_{i=1}^{n} appli_i)$) et toutes les applications impliquées dans la composition d'IHM. A partir de ces informations et des définitions 4 and 7, toutes les entrées, sorties et actions du composant composite fonctionnel doivent être liées aux composants d'IHM résultants.

Propriété 7 (Ports des composants d'IHM résultants)**.** *Les propriétés suivantes concernent les ports des composants d'IHM qui résultent de la fonction de composition des IHM. On appelle ici caui un des composants d'IHM présent dans l'ensemble des composants d'IHM obtenu par la fonction* $app\_caui(appli_{composed})$ *et* cfc_{res} *le composant composite fonctionnel qui est le résultat de la fonction* $app\_cfc(appli_{composed})$*. La fonction* $copy\_of$ *crée une copie de l'élément sélectionné.*

$$caui\_inputs(caui) = \left\{ i = copy\_of(i_3) \mid \left(\begin{array}{l} \forall i_1 \in cfc\_inputs(cfc_{res}), \exists i_2 \mid \\ (i_1, i_2) \in toscdlinkII(cfc_{res}, cfc_1) \wedge \\ (i_3, i_2) \in dataLinkII(caui_1, cfc_1) \Rightarrow \\ dataLinkII(caui, cfc_{res}) = \\ dataLinkII(caui, cfc_{res}) \cup \{(i, i_1)\} \end{array} \right) \right\} \quad (4.34)$$

$$caui\_outputs(caui) = \left\{ o = copy\_of(o_3) \mid \left(\begin{array}{l} \forall o_1 \in cfc\_outputs(cfc_{res}), \exists o_2 \mid \\ (o_2, o_1) \in toscdlinkOO(cfc_1, cfc_{res}) \wedge \\ (o_3, o_2) \in dataLinkOO(caui_1, cfc_1) \Rightarrow \\ dataLinkOO(caui, cfc_{res}) = \\ dataLinkOO(caui, cfc_{res}) \cup \{(o, o_1)\} \end{array} \right) \right\} \quad (4.35)$$

$$caui\_events(caui) = \left\{ e = copy\_of(e_3) \mid \left(\begin{array}{l} \forall a_1 \in cfc\_actions(cfc_{res}), \exists a_2 \mid \\ (a_1, a_2) \in toscalink(cfc_{res}, cfc_1) \wedge \\ (e_1, a_2) \in eventLink(caui_1, cfc_1) \Rightarrow \\ eventLink(caui, cfc_{res}) = \\ eventLink(caui, cfc_{res}) \cup \{(e, a_1)\} \end{array} \right) \right\} \quad (4.36)$$

Avec :

- $cfc_1 = app\_cfc(appli_1)$, *un des composants fonctionnels d'une des applications impliquées dans la composition* ($appli_1 \in \cup_{i=1}^{n} appli_i$)
- $caui_1 \in app\_caui(appli_1)$, *un des composants d'IHM associé au composant fonctionnel* cfc_1 *présent dans la même application* $appli_1$

Exemple 7 (Ports d'IHM sélectionnés pour le composant d'IHM)**.** On illustre ici les ports d'IHM qui se trouvent dans le composant d'IHM résultant de la composition. Ce sont les ports qui ont été sélectionnés et placés temporairement dans le composant d'IHM et liés avec le composant composite fonctionnel. Par la suite, en cas de conflits, ces ports peuvent être modifiés. De la composition, un seul composant d'IHM est créé. On nomme se composant d'IHM : $caui_{res}$.

$$\begin{aligned} caui\_inputs(caui_{res}) &= \{copy\_input_1, copy\_input_2\} \\ caui\_outputs(caui_{res}) &= \{copy\_output_1, copy\_output_2\} \\ caui\_events(caui_{res}) &= \{copy\_event_1, copy\_event_2\} \end{aligned}$$

Avec :

- $copy\_input_1$ qui correspond à la copie de l'entrée $input_1$ du composant d'IHM $caui_{getByCard}$, c'est-à-dire, l'élément d'IHM qui permet la saisie du numéro de carte de sécurité sociale ;
- $copy\_input_2$ qui correspond à la copie de l'entrée $input_1$ du composant d'IHM $caui_{getAddresses}$, c'est-à-dire, l'élément d'IHM qui permet la saisie du nom complet ;
- $copy\_output_1$ qui correspond à la copie de la sortie $output_1$ du composant d'IHM $caui_{getByCard}$, c'est-à-dire, l'élément d'IHM qui permet l'affichage de l'adresse personnelle ;
- $copy\_output_2$ qui correspond à la copie de l'entrée $output_1$ du composant d'IHM $caui_{getAddresses}$, c'est-à-dire, l'élément d'IHM qui permet l'affichage de l'ensemble des adresses professionnelles ;
- $copy\_event_1$ qui correspond à la copie de l'entrée $event_1$ du composant d'IHM $caui_{getByCard}$, c'est-à-dire, l'élément d'IHM qui permet le déclenchement de l'appel de l'opération *getByCard* ;
- $copy\_event_2$ qui correspond à la copie de l'entrée $event_1$ du composant d'IHM $caui_{getAddresses}$, c'est-à-dire, l'élément d'IHM qui permet le déclenchement de l'appel de l'opération *getAddresses* ;

Seconde étape : détection des points de conflits

Durant le processus de composition et après avoir sélectionné les éléments d'IHM qui doivent être conservés, on réalise une étape de détection de conflits. Des conflits peuvent apparaître lorsqu'une entrée du composant composite fonctionnel est partagé par plusieurs entrées de ses sous-composants fonctionnels ou bien lorsque plusieurs sorties des sous-composants fonctionnels sont fusionnées pour n'en obtenir qu'une seule. On détecte quatre types de conflits qui sont :

- Les conflits de dénomination lorsque l'on a deux labels associés à un port différent. Pour ce faire, on vérifie l'attribut *name* des ports en question. Ce conflit peut apparaître sur des entrées, des sorties et des événements. Ce conflit est illustré dans l'exemple 8.
- Les conflits de types lorsque l'on a deux types différents (*e.g. string* et *number*). Ce conflit peut apparaître sur des entrées et sorties.
- Les conflits de cardinalité lorsque l'on a deux cardinalités différentes (*e.g.* 1 et *n* ce qui signifie que l'on a un élément dédié pour un afficher ou saisir une valeur et un autre pour afficher ou saisir un ensemble de valeurs). Ce conflit peut apparaître sur des entrées et sorties. Il est illustré dans l'exemple 8.
- Les conflits de sélection lorsque l'on a deux sélections différentes (*e.g. single* et *multiple*). Ce conflit ne peut apparaître que sur des entrées.

Avant de présenter les fonctions de détection de conflits, on introduit la notion d'égalité entre élément d'IHM. Ainsi, pour que deux ports soient égaux, il est nécessaire que l'ensemble de ses attributs soient égaux (cf. définition 12).

Définition 12 (Egalité de ports de composant d'IHM)**.** L'égalité entre deux ports de composants d'IHM est définie par :

$$in_1 = in_2 \Rightarrow \begin{cases} caui\_in\_name(in_1) = caui\_in\_name(in_2) \wedge \\ caui\_in\_uitype(in_1) = caui\_in\_uitype(in_2) \wedge \\ caui\_in\_cardinality(in_1) = caui\_in\_cardinality(in_2) \wedge \\ caui\_in\_selection(in_1) = caui\_in\_selection(in_2) \end{cases} \tag{4.37}$$

$$out_1 = out_2 \Rightarrow \begin{cases} caui\_out\_name(out_1) = caui\_out\_name(out_2) \wedge \\ caui\_out\_uitype(out_1) = caui\_out\_uitype(out_2) \wedge \\ caui\_out\_cardinality(out_1) = caui\_out\_cardinality(out_2) \end{cases} \tag{4.38}$$

$$evt_1 = evt_2 \Rightarrow event\_name(evt_1) = event\_name(evt_2) \tag{4.39}$$

La détection des points de conflits (cf. définition 13) se fait au travers de trois fonctions : $ConflictIn$, $ConflictOut$ et $ConflictEvent$. Chacune des ces fonctions associe à un port du composant composite fonctionnel l'ensemble des éléments qui sont en conflits. Pour qu'il y ait conflit, il est nécessaire qu'au moins deux éléments d'IHM diffèrent.

Définition 13 (Conflit)**.** Les trois fonctions qui suivent permettent de détecter l'ensemble des points de conflits associés à chacun des ports du composant composite fonctionnel. On appelle cfc_{res} le composant composite fonctionnel et *caui* un des composants d'IHM présent dans l'ensemble des composants d'IHM obtenu par la fonction $app\_caui(appli_{composed})$.

$$ConflictIn : CAUI \times INPUT_{cfc} \rightarrow INPUT^*_{caui} \tag{4.40}$$

$$i \in cfc\_inputs(cfc_{res}), conflictIn(caui, i) = \left\{ \begin{array}{c} in \mid (in, i) \in \\ dataLinkII(caui, cfc_{res}) \end{array} \right\} \tag{4.41}$$

$$ConflictOut : CAUI \times OUTPUT_{cfc} \rightarrow OUTPUT^*_{caui} \tag{4.42}$$

$$o \in cfc\_outputs(cfc_{res}), conflictOut(caui, o) = \left\{ \begin{array}{c} out \mid (out, o) \in \\ dataLinkOO(caui, cfc_{res}) \end{array} \right\} \tag{4.43}$$

$$ConflictEvent : CAUI \times ACTION \rightarrow EVENT^* \tag{4.44}$$

$$a \in cfc\_actions(cfc_{res}), conflictEvent(caui, a) = \left\{ \begin{array}{c} evt \mid (evt, a) \in \\ eventLink(caui, cfc_{res}) \end{array} \right\} \tag{4.45}$$

$$\tag{4.46}$$

Propriété 8 (Propriété sur les conflits)**.** *Pour qu'il y ait un conflit, il est nécessaire que deux ports d'IHM diffèrent, c'est-à-dire qu'ils ne sont pas égaux par rapport à la définition 12. On appelle cfc_{res} le composant composite fonctionnel et caui un des composants d'IHM présent dans l'ensemble des composants d'IHM obtenu par la fonction $app\_caui(appli_{composed})$.*

$$\exists i \in cfc\_inputs(cfc_{res}), conflictIn(caui, i) \Rightarrow \begin{cases} card(conflictIn(caui, i)) \geq 2 \wedge \\ \left(\begin{array}{l} \exists in_1, \exists in_2 \in \\ conflictIn(caui, i) \mid \\ in_1 \neq in_2 \end{array} \right) \end{cases}$$

$$\exists o \in cfc\_outputs(cfc_{res}), conflictOut(caui, o) \Rightarrow \begin{cases} card(conflictOut(caui, o)) \geq 2 \wedge \\ \left(\begin{array}{l} \exists out_1, \exists out_2 \in \\ conflictOut(caui, o) \mid \\ out_1 \neq out_2 \end{array} \right) \end{cases}$$

$$\exists a \in cfc\_actions(cfc_{res}), conflictEvent(caui, a) \Rightarrow \begin{cases} card(conflictEvent(caui, a)) \geq 2 \wedge \\ \left(\begin{array}{l} \exists evt_1, \exists evt_2 \in \\ conflictEvent(caui, a) \mid \\ evt_1 \neq evt_2 \end{array} \right) \end{cases}$$

Sur l'exemple, on va détecter deux conflits : un conflit sur les sorties et un conflit sur les événements (cf. exemple 8). En effet, le nom associé aux deux événements diffère et pour les sorties, le nom et la cardinalité diffèrent.

Exemple 8 (Conflits sur les sorties et les événements)**.** On illustre ici les conflits que l'on détecte et qui sont présents au sein de la composition.

$$\begin{aligned} conflictOut(caui_{res}, coutput_1) &= \{copy\_output_1, copy\_output_2\} \\ caui\_out\_name(copy\_output_1) &\neq caui\_out\_name(copy\_output_2) \\ address &\neq addresses \\ caui\_out\_cardinality(copy\_output_1) &\neq caui\_out\_cardinality(copy\_output_2) \end{aligned}$$

Les éléments d'IHM associés à la sortie de la composition fonctionnelle sont en conflit à la fois sur le nom et sur la cardinalité.

$$\begin{aligned} conflictEvent(caui_{res}, caction_1) &= \{copy\_event_1, copy\_event_2\} \\ event\_name(copy\_event_1) &\neq event\_name(copy\_event_2) \\ getByCard &\neq getAddresses \end{aligned}$$

Les éléments d'IHM associés à l'action sont en conflit sur le nom.

On fournit également une fonction de suppression automatique d'éléments d'IHM inconsistants présents au sein des conflits. L'inconsistance apparaît :

- lorsque des sorties ont été fusionnées au sein de la composition fonctionnelle et que deux cardinalités sont différentes (*e.g.* $[0,1]$ et $[0,n]$ qui donnent $[0,n]$) et que l'on retrouve cela au niveau des éléments d'IHM (*e.g.* 1 et n).
- lorsque des sorties ont été fusionnées au sein de la composition fonctionnelle pour former une cardinalité plus grande que celles des éléments de bases (*i.e.* deux éléments avec une cardinalité de $[0,1]$ qui donnent $[0,n]$)
- lorsqu'une entrée est partagée par plusieurs entrées de sous-composants, que deux cardinalités diffèrent et que l'on conserve la plus petite.

Pour permettre de supprimer les éléments inconsistants, on utilise une table (cf. table 4.1) qui pour une cardinalité d'un port d'un composant fonctionnel fait correspondre la cardinalité possible à connecter dessus pour un port d'un composant d'IHM. La fonction de suppression des éléments inconsistants n'est applicable que dans le cas où il n'y a qu'un conflit de cardinalité. Si aucune cardinalité ne convient, celles-ci sont conservées et présentées comme un conflit.

	Cardinalité du port du composant fonctionnel	Cardinalité du port du composant d'IHM
Port d'entrée	$[0,1]$ ou $[1,1]$	1 ou n (si $selection = single$)
	$[0,n]$ ou $[1,n]$	1 ou n
Port de sortie	$[0,1]$ ou $[1,1]$	1 ou n
	$[0,n]$ ou $[1,n]$	n

TABLE 4.1 – Table de consistence qui fait correspondre à chaque port du fonctionnel la cardinalité possible pour le port d'IHM en fonction du type de port

Troisième étape : résolution des conflits

Il existe quatre conflits différents à résoudre qui sont :
- les conflits de dénomination. Les éléments graphiques ne possèdent pas le même nom et donc il est nécessaire de déterminer quel est le nom qui doit être conservé.
- les conflits de types. Les éléments graphiques possèdent des types différents. Il est alors intéressant de prendre celui qui peut convenir le mieux pour l'usage du composant composite.
- les conflits de cardinalité. Si les éléments n'ont pas la même cardinalité, il faut décider si l'on souhaite conserver une liste ou un élément unaire. Ceux-ci restent à résoudre dans le cas où ils sont associés à d'autres conflits et qu'ils n'ont pas été automatiquement résolus.
- les conflits de sélection. Lorsque deux éléments représentent des listes mais qu'ils ne possèdent pas la même liberté de sélection, lequel doit alors être conservée.

La résolution des conflits est effectuée de manière manuelle par le développeur qui est en train de réaliser la composition. Plusieurs choix lui sont offerts. En effet, il peut :
- choisir une représentation graphique parmi l'ensemble des représentations présentes ;
- choisir de conserver l'ensemble des représentations graphiques présentes ce qui implique que toutes les éléments d'IHM doivent présenter la même information et qu'il est donc nécessaire de mettre en place un mécanisme de mise à jour ;
- sélectionner des informations en provenance de différentes représentations graphiques permet ainsi de mélanger les caractéristiques de plusieurs représentations graphiques ;
- personnaliser complètement le résultat en fournissant lui-même les informations pour obtenir une nouvelle représentation graphique.

L'objectif à la fin de l'étape de résolution de conflits est de compléter la liste des éléments sans conflits afin de finaliser le processus de composition et d'ainsi pouvoir créer l'ensemble des composants d'IHM associés à la composition fonctionnelle.

Pour aider le développeur dans la résolution des conflits, un ensemble d'informations lui est fourni. Ces informations présentent au développeur le port du composant fonctionnel sur lequel il y a un conflit, sur quelles informations se trouvent le conflit et enfin les différentes possibilités offertes pour résoudre le conflit. Ces informations sont nécessaires pour une meilleure réutilisation des informations déjà présentes.

Afin de laisser une plus grande liberté au développeur sur la résolution des conflits, le choix de la politique de résolution de conflit est effectué sur chacun des ports qui possèdent un conflit.

A la fin de cette étape on obtient, pour chaque élément en conflits, le choix effectué par le développeur (cf. définition 14). Pour chaque résolution, on peut avoir les deux données suivantes : une liste (qui correspond aux éléments à supprimer) et un élément (qui correspond à l'élément que l'on crée). Il y a alors trois possibilités : (i) on conserve tous les éléments, dans ce cas là, la liste est vide et il n'y a pas d'élément ; (ii) on ne conserve qu'un élément, dans ce cas là, la liste contient l'ensemble des éléments qui doivent être supprimés (*i.e.* la liste initiale des éléments où l'on a retiré l'élément que l'on souhaite conserver) et il n'y a pas d'élément ; et (iii) on crée un nouvel élément dans ce cas là, la liste contient l'ensemble des éléments qui doivent être retiré et l'élément correspond à celui qui vient en remplacement des autres.

Définition 14 (Conflits résolus)**.** Les trois fonctions qui suivent fournissent pour chaque conflit le choix effectué par le développeur. On appelle cfc_{res} le composant composite fonctionnel et *caui* un des composants d'IHM présent dans l'ensemble des composants d'IHM obtenu par la fonction $caui(appli_{composed})$. Le premier élément du couple du résultat de la fonction correspond à l'ensemble des éléments conservés, le second élément correspond à une création.

$$ResolvedConflictIn : CAUI \times IN_{cfc} \rightarrow IN^*_{caui} \times IN_{caui} \quad (4.47)$$
$$\forall i \in cfc\_inputs(cfc_{res}) \mid \exists ConflictIn(caui, i),$$

$$resolvedConflictIn(caui, i) = \begin{cases} (\{\}, \varnothing) \ (i) \\ (conflictIn(caui, i) - in_{caui}, \varnothing) \ (ii) \\ (conflictIn(caui, i), new\_in_{caui}) \ (iii) \end{cases} \quad (4.48)$$

$$ResolvedConflictOut : CAUI \times OUT_{cfc} \rightarrow OUT^*_{caui} \times OUT_{caui} \quad (4.49)$$
$$\forall o \in cfc\_outputs(cfc_{res}) \mid \exists ConflictOut(caui, o),$$

$$resolvedConflictOut(caui, o) = \begin{cases} (\{\}, \varnothing) \ (i) \\ (conflictOut(caui, o) - out_{caui}, \varnothing) \ (ii) \\ (conflictOut(caui, o), new\_out_{caui}) \ (iii) \end{cases} \quad (4.50)$$

$$ResolvedConflictEvent : CAUI \times ACTION \rightarrow EVENT^* \times EVENT \quad (4.51)$$
$$\forall a \in cfc\_actions(cfc_{res}) \mid \exists ConflictEvent(caui, a),$$

$$resolvedConflictEvent(caui, a) = \begin{cases} (\{\}, \varnothing) \ (i) \\ (conflictEvent(caui, a) - evt_{caui}, \varnothing) \ (ii) \\ (conflictEvent(caui, a), new\_evt_{caui}) \ (iii) \end{cases} \quad (4.52)$$

Dans le cadre de l'exemple, le développeur doit résoudre deux conflits (cf. exemple 9). Dans le cas du conflit sur l'événement, il décide de créer un nouveau bouton avec pour nom : *getAllAddresses*. Dans le cas du conflit sur la sortie, il décide de conserver l'élément $copy\_output_2$ qui a pour nom *addresses* et pour cardinalité n.

Exemple 9 (Conflits résolus par le développeur)**.** On illustre ici la résolution des conflits faite par le développeur.

$$\begin{aligned} ResolvedConflictOut(caui_{res}, coutput_1) &= (ConflictOut(caui_{res}, coutput_1) - copy\_output_2, \varnothing) \\ ResolvedConflictEvent(caui_{res}, caction_1) &= (ConflictEvent(caui_{res}, caction_1), new\_event_1) \\ event\_name(new\_event_1) &= getAllAddresses \end{aligned}$$

Quatrième étape : Finalisation de la composition et création de l'application

La création des composants d'IHM est réalisée en se basant sur l'ensemble des informations précédemment obtenues. On utilise les fonctions définies en 14 afin de résoudre les conflits. Selon

les éléments présents, on conserve tous les éléments d'IHM, on n'en conserve qu'un ou bien on supprime tous les éléments d'IHM précédemment sélectionnés pour les remplacer par un nouvel élément. Dans le cas où l'on conserve l'ensemble des éléments, il est nécessaire d'introduire des liens internes à l'IHM pour permettre d'assurer que l'ensemble des éléments possède bien le même contenu (cf. définition 15). En plus de ces liens, on complète également les événements présents dans les composants d'IHM afin que ceux-ci soient liés à leurs ports d'entrée et de sortie. Le résultat obtenu est conforme à la définition qui a été faite d'une application (cf. définition 1).

Définition 15 (Liens internes de cohérence)**.** Les liens internes de cohérence relient les ports d'IHM entre eux pour assurer que tous les éléments possèdent la même valeur. Ainsi, dans le cas des entrées, ils permettent d'assurer que l'ensemble des couples i_1, i_2 possède la même valeur saisie par l'utilisateur et dans les cas des sorties que l'ensemble des couples o_1, o_2 affiche la même valeur. Ils sont définis par :

$$InternalLink : CAUI \rightarrow caui\_inputq(CAUI) \times caui\_inputq(CAUI) \tag{4.53}$$

$$\forall caui \in app\_caui(appli_{composed}),$$

$$internalLink(caui) = \left\{ \begin{array}{l} (i_1, i_2) \mid \exists i \in cfc\_input(cfc_{res}), \\ (listReused, newElem) = resolvedConflictIn(caui, i) \wedge \\ i_1 \in listReused \wedge i_2 \in listReused \end{array} \right\} \tag{4.54}$$

$$InternalLink : CAUI \rightarrow caui\_outputs(CAUI) \times caui\_outputs(CAUI) \tag{4.55}$$

$$\forall caui \in app\_caui(appli_{composed}),$$

$$internalLink(caui) = \left\{ \begin{array}{l} (o_1, o_2) \mid \exists o \in cfc\_output(cfc_{res}), \\ (listReused, newElem) = resolvedConflictIn(caui, i) \wedge \\ o_1 \in listReused \wedge o_2 \in listReused \end{array} \right\} \tag{4.56}$$

4.2.4 Synthèse et apport de la formalisation et méta-modélisation

Dans cette section, nous avons vu la formalisation et la méta-modélisation de l'ensemble des éléments d'entrée et de sortie du processus de composition. Ces deux représentations sont impliquées dans deux étapes différentes du processus. En effet, la formalisation est utilisée au sein du moteur de composition dans les algorithmes qui permettent de déduire les éléments d'IHM à conserver et à fusionner. L'ensemble de la formalisation a été implémentée en `Prolog` au sein d'un atelier d'aide à la composition présenté dans le chapitre 5 en section 5.2 et dont le code se trouve en annexe B. La méta-modèlisation est quant à elle utilisée dans l'ensemble des mécanismes de transformations qui permettent de réaliser l'abstraction et la concrétisation mais également l'interaction avec le moteur de composition. Elle est validée sur l'ensemble des transformations d'abstraction et de concrétisation ainsi que sur les contraintes OCL qui correspondent aux besoins exprimés dans le méta-modèle. Ces transformations sont présentées dans le chapitre 5 en section 5.1.

Les deux représentations présentées dans ce chapitre définissent ce qu'est une application au travers des composants fonctionnels et des composants d'IHM et des liens qui les unissent. Elles définissent également la composition fonctionnelle grâce à une représentation sous forme de composant composite et de l'ensemble des liens qui relient celui-ci à ses sous-composants. Enfin elles définissent ce à quoi doit correspondre le résultat de la composition d'IHM. Les définitions présentes dans la composition d'IHM qui correspondent à différentes étapes de la réalisation de cette composition se retrouve au sein des règles `Prolog`. Ainsi, pour citer les principales, on a : pour la définition 13 (Points de conflits) et 12 (Egalité d'éléments d'IHM) , les règles `Prolog` `conflict_detection_input` (resp. `out` et `event`) qui utilisent la règle `test` qui teste les différentes composantes d'une entrée (resp. sortie et event) afin de vérifier que les éléments sont identiques ou pas et donc de détecter les points de conflits. Les autres règles `Prolog` se trouvent en annexe B dans les différentes étapes correspondantes au processus du moteur de composition d'IHM.

L'intérêt de ces deux représentations réside dans les propriétés mises en œuvre qui permettent de réaliser la composition d'IHM et de déduire les éléments d'IHM qui doivent être conservés en fonction de la composition fonctionnelle et des applications impliquées dans la composition fonctionnelle. Ces propriétés permettent de contraindre les représentations que l'on a de chacun des éléments nécessaires à la réalisation de la composition et elles permettent également de contrôler le résultat obtenu. Les propriétés que l'on a sur une application sont : que les éléments présents dans un composant d'IHM sont tous reliés au composant fonctionnel (on restreint la représentation de l'IHM aux éléments qui sont utilisés par le composant fonctionnel (cf. propriété 1) et que tous les éléments (entrées et sorties) requis par une opération utilisée sont connectés à des éléments d'un composant d'IHM (cf. propriétés 1 et 2). La propriété la plus importante sur la composition fonctionnelle est celle concernant la vérification de l'intégrité de celle-ci (cf. propriété 6).

4.3 Conclusion

Ce chapitre a présenté l'ensemble de la démarche mise en œuvre pour réaliser la composition d'applications. Ce processus s'appuie sur quatre étapes que sont la description des applications existantes à composer, la description de la composition fonctionnelle, la réalisation de la composition d'IHM et pour finir la concrétisation du résultat. La méta-modélisation des applications et de leur composition fonctionnelle met en relief le découpage Noyau Fonctionnel–IHM et les liens qui relient le noyau fonctionnel à l'IHM comme préconisé dans les architectures de type MVC, PAC ou Arch. La représentation choisie est une représentation à base de composants : un composant fonctionnel, un ensemble de composants d'IHM et des liens de données et d'événements entre le composant fonctionnel et les composants d'IHM. Chaque IHM est associée à une ou plusieurs opérations (*action*) présentes au sein du composant fonctionnel. Les composants d'IHM ne sont associés qu'à un seul composant fonctionnel. Le choix a également été fait de distinguer à la fois des ports de données et des ports d'événement/action afin de dissocier le flot de contrôle et le flot de données que l'on retrouve dans les orchestrations de services par exemple (cf. 3.1.1). La représentation qui a été faite de la composition fonctionnelle correspond à un composant composite qui agrège l'ensemble des composants fonctionnels impliqués dans la composition. De la même manière on distingue des ports de données et d'action et on retrouve les deux types de liens que sont les liens de données et d'action.

La composition d'IHM déduite par le processus de composition utilise les informations extraites de la composition fonctionnelle pour identifier les éléments d'IHM qui doivent être conservés ou fusionnés. De ce fait, des choix qui pouvaient apparaître, en ne s'intéressant qu'aux IHM (cf. 3.2.3), disparaissent grâce au modèle de tâches implémenté au niveau des orchestrations. Associé à une bonne connaissance des interactions NF/IHM, les applications résultantes peuvent inclure des fusions et partages de données au niveau de leur IHM ce qui est plus complet que les approches qui utilisent la composition fonctionnelles pour réaliser la composition d'applications (cf. 3.3.2). De plus, les mécanismes de composition permettent, en cas de conflits, d'aider le développeur en fournissant des informations précises sur l'usage qui était fait au préalable des éléments d'IHM en concurrence.

La formalisation est au cœur de l'implémentation du moteur de composition réalisé en `Prolog` et qui respecte l'ensemble des définitions, propriétés et étapes de la composition d'IHM présenté en section 4.2.3. Ce moteur de composition a été intégré au sein d'un atelier d'aide à la composition afin de valider l'approche. L'ensemble de l'implémentation du moteur de composition se trouve en annexe B. La méta-modélisation a quant à elle été utilisée au sein des transformations d'abstraction. Ces transformations permettent ainsi d'obtenir, à partir d'un *Web Service*, d'une orchestration de *Web Services* ou d'IHM décrites en `Flex` une instance du méta-modèle `AliasComponent` conforme aux contraintes présentes sur celui-ci.

La chapitre 5 présente la validation de l'approche avec dans un premier temps les transformations qui ont été mises en place (cf. section 5.1) puis l'évaluation de l'atelier d'aide à la composition (cf. section 5.2).

5 Validation de l'approche Alias

Sommaire

5.1 Les transformations au sein du processus de composition 120
5.1.1 Abstraction des éléments nécessaires à la réalisation de la composition . . . 120
5.1.2 Concrétisation du résultat obtenu pour permettre la validation 132
5.1.3 Conclusion 133
5.2 Atelier d'aide à la composition d'IHM 134
5.2.1 Présentation de l'atelier 134
5.2.2 Tests utilisateurs réalisés sur l'atelier 136
5.3 Conclusion 142

Ce chapitre a pour objectif de valider la démarche exposée dans le chapitre 4. C'est-à-dire valider le méta-modèle au centre de la procédure de composition ainsi que la formalisation. Le processus de composition et plus particulièrement la détection et résolution de conflits ont été validés par la mise en œuvre de tests utilisateurs.

La validation des méta-modèles passe par l'écriture de transformations (cf. section 5.1). Celles-ci permettent de réaliser l'abstraction des applications à composer ainsi que de la composition fonctionnelle ; et la concrétisation du résultat obtenu par le moteur de composition. L'obtention du méta-modèle `AliasComponent` est passée par la réalisation de plusieurs méta-modèles intermédiaires. Les premiers méta-modèles mis en place se sont focalisés sur la description des IHM dans leur ensemble, c'est-à-dire, décrire la structure de l'IHM (`AliasBehavior`), mais également les éléments qui les composent (`AliasStructure`) et pour finir le placement de ces éléments (`AliasLayout`) [PDJR+08]. Pour la réalisation de la composition, il est nécessaire de décrire en plus de l'IHM, le noyau fonctionnel et les interactions qui existent entre l'IHM et le noyau fonctionnel ainsi que la composition fonctionnelle. C'est pourquoi, nous nous sommes focalisés sur le méta-modèle de description structurelle de l'IHM (`AliasBehavior` est devenu `AliasExchange` car il permettait l'échange des descriptions des IHM entre les développeurs) que l'on a complété par un second méta-modèle (`AliasCompose`) décrivant les interactions et les assemblages effectués au niveau fonctionnel [DPJO+09, OJDP+10b, OJDP10a]. Finalement, ces deux méta-modèles ont été fusionnés afin d'obtenir l'actuel `AliasComponent`.

La section 5.2 présente l'atelier qui a été réalisé et qui permet de résoudre la composition d'IHM en s'appuyant sur la formalisation décrite en section 4.2. Cet atelier fournit également l'ensemble des informations nécessaires à la résolution des conflits afin d'aider le développeur dans le processus de composition. L'évaluation de cet atelier se trouve également dans cette section. Cette évaluation a permis de valider la détection et la résolution de conflits proposées dans le chapitre 4.

5.1 Les transformations au sein du processus de composition

Les transformations présentes au sein du framework permettent de réutiliser l'ensemble des informations existantes que ce soit au niveau des applications à composer ou au niveau de la composition fonctionnelle. D'autres transformations réalisent la concrétisation du résultat obtenu par le processus de composition pour obtenir une visualisation. Les transformations d'abstraction sont de type *model-to-model* selon la classification de Czarnecki *et al.* [CH03] alors que la transformation de concrétisation est de type *model-to-code*.

La sous-section 5.1.1 présente l'ensemble des transformations qui permettent d'abstraire une IHM, la partie fonctionnelle ainsi que la composition fonctionnelle. Elle présente également des possibilités d'analyse pour la partie interaction.

La sous-section 5.1.2 présente la concrétisation de l'IHM afin d'obtenir un visualisation du résultat obtenu après composition.

5.1.1 Abstraction des éléments nécessaires à la réalisation de la composition

Plusieurs éléments doivent être abstraits pour permettre la réalisation de la composition d'IHM. Cette abstraction permet également d'avoir une représentation homogène de l'ensemble des éléments manipulés en étant indépendante d'une technologie donnée. Les sous-sections qui suivent présentent l'ensemble des abstractions, c'est-à-dire, l'abstraction de la partie fonctionnelle, de la partie IHM, de l'interaction et la composition fonctionnelle. L'ensemble des éléments abstraits est décrit dans le méta-modèle `AliasComponent`.

Abstraction de la partie fonctionnelle d'une application

L'abstraction de la partie fonctionnelle consiste en la création d'un composant fonctionnel (*FCComponent*) du méta-modèle `AliasComponent`. Ce composant a pour *Actions* l'ensemble des méthodes/opérations disponibles au sein du composant ou de service à abstraire. Les éléments d'entrée et sortie associés à une *Action* correspondent aux paramètres d'entrée et de sortie d'une méthode/opération.

La réalisation de cette abstraction s'appuie sur une transformation `ATL` [JAB+06, JK05]. On transforme ainsi un modèle WSDL en un modèle `AliasComponent`. L'intérêt de cette transformation est de garantir la conformité des modèles à leurs méta-modèles correspondants. Pour ce faire, il a été nécessaire de décrire les deux méta-modèles (WSDL et `AliasComponent`) au sein d'`eCore` (cf. annexe C). Ces méta-modèles `eCore` sont utilisés dans les règles de transformations d'`ATL`.

La transformation permet de réaliser un *mapping* entre des éléments du WSDL et des éléments du composant fonctionnel. Ainsi des opérations sont transformées en action et les messages d'entrée et de sortie associés à chacune des actions sont transformés en entrées et en sorties.

Pour permettre cette transformation, des *helpers*, propres à `ATL`, ont été créés. Ils fournissent un ensemble de fonctions permettant :

– d'identifier de manière unique les ports créés. Pour ce faire, on utilise des compteurs que l'on incrémente à chaque fois que l'on ajoute un nouveau port. Trois compteurs ont été mis en place :*counterA*, *counterI* et *counterO* pour les trois types de ports qui existent ;

– de faire correspondre à chaque élément complexe l'ensemble des éléments qui le compose. Cette correspondance est faite par la fonction : *mapComplex2Sequence ;*
– d'accéder à un des ensembles créés par la fonction précédente à partir du nom du type complexe. Cela est fait par la fonction : *retrieveSequenceFromComplex*
– de créer les listes correspondantes aux ports d'entrée et de sortie. Ces deux listes sont utilisées une fois que l'ensemble des ports d'action ont été créé afin d'ajouter au composant fonctionnel l'ensemble de ses ports d'entrée et de sortie. Ceci est fait par les deux fonction : *inputP* et *outputP*.

Le listing 5.1 fournit la définition de l'ensemble des *helpers*.

```
helper def: counterA : Integer = 0;
helper def: counterI : Integer = 0;
helper def: counterO : Integer = 0;
helper def: inputP : Sequence(AliasComponent!Input) = Sequence{};
helper def: outputP : Sequence(AliasComponent!Output) = Sequence{};

helper def: mapComplex2Sequence : Map(String,Sequence(WSDL!Element)) =
   WSDL!ComplexType.allInstances()->
      iterate(e; ret : Map(String, Sequence(WSDL!Element)) = Map{}|
         ret->including(e.Name, e.element)
      );

helper def: retrieveSequenceFromComplex (elm : WSDL!ComplexType) : Sequence(WSDL!Element) =
   thisModule.mapComplex2Sequence.get(elm.Name);
```

Listing 5.1 – *helpers* mis en place pour aider à la transformation

La règle principale `WSDL2AliasComponent` qui est appelée pour permettre de réaliser la transformation a pour entrée l'élément *Definition* du WSDL et comme sortie un *FCComponent* d'`AliasComponent`. Cette règle permet de créer l'ensemble des ports du composant fonctionnel. Elle fait appel à d'autres règles qui créent chacun des types de port. Le listing 5.2 fournit cette règle. Les listes définies dans les *helpers* sont utilisées ici pour créer les ports de données d'entrée et sortie.

```
rule WSDL2AliasComponent {
   from
      wsdl : WSDL!Definition
   to
      out : AliasComponent!FCComponent(
         id <- wsdl.Name,
        action <- WSDL!Operation->allInstances()->
           collect(e | thisModule.Operation2Action(e)),
        input <- thisModule.inputP,
        output <- thisModule.outputP
      )
}
```

Listing 5.2 – Règle principale qui permet le déclenchement de la transformation

La règle `WSDL2AliasComponent` fait appel à une autre règle qui crée l'ensemble des actions à partir des opérations présentes dans le WSDL (`Operation2Action`). Cette règle qui crée les actions va faire appel à deux autres règles pour créer les ports d'entrées et de sorties. On utilise ici le compteur spécifique aux actions afin d'assurer l'unicité des identifiants. Le listing 5.3 correspond à la règle qui transforme une opération en action.

```
lazy rule Operation2Action {
   from
      op : WSDL!Operation
   using{
      ct : Integer = thisModule.counterA;
   }
   to
      a : AliasComponent!Action (
        name<-op.Name,
        id<-'action_'+ct,
        inputs <- if op.input.message.message.first().type.oclIsUndefined()
          then if op.input.message.message.first().element.type.oclIsTypeOf(WSDL!ComplexType)
```

```
            then thisModule.retrieveSequenceFromComplex(
              op.input.message.message.first().element.type) ->
                collect(e | thisModule.Element2Input(e))
            else thisModule.Element2Input(op.input.message.message.first().element)
            endif
          else if op.input.message.message.first().type.oclIsTypeOf(WSDL!ComplexType)
            then thisModule.retrieveSequenceFromComplex(
              op.input.message.message.first().type) ->
                collect(e | thisModule.Element2Input(e))
            else thisModule.SimpleType2Input(
              op.input.message.message.first().Name,op.input.message.message.first().type)
            endif
          endif,
          outputs <- if op.output.message.message.first().type.oclIsUndefined()
            then if op.output.message.message.first().element.type.oclIsTypeOf(WSDL!ComplexType)
              then thisModule.retrieveSequenceFromComplex(
                op.output.message.message.first().element.type) ->
                  collect(f | thisModule.Element2Output(f))
              else thisModule.Element2Output(op.output.message.message.first().element)
              endif
            else if op.output.message.message.first().type.oclIsTypeOf(WSDL!ComplexType)
              then thisModule.retrieveSequenceFromComplex(
                op.output.message.message.first().type) ->
                  collect(f | thisModule.Element2Output(f))
              else thisModule.SimpleType2Output(
                op.output.message.message.first().Name,op.output.message.message.first().type)
              endif
            endif)
    do{
      thisModule.counterA <- thisModule.counterA + 1;
    }
}
```

Listing 5.3 – Règle principale qui permet le déclenchement de la transformation

Enfin la règle `Element2Input` (resp. `Element2Output`) permet de transformer un élément en entrée (resp. en sortie). Des tests sont effectués sur les valeurs *MaxOccurs* et *MinOccurs* afin de déterminer la bonne cardinalité. Ainsi les cardinalités fixées suivent les contraintes OCL fournies par le listing 4.1. On se sert également de deux *helpers* dans cette règle pour gérer le compteur et également ajouter le port créé à la liste des ports qui est utilisée dans la création du composant fonctionnel. Le listing 5.4 fournit la définition de la règle `Element2Input` (la règle `Element2Output` étant similaire).

```
lazy rule SimpleType2Input{
    from
        nameE : String ,
        typeEl : String
    using{
        ct : Integer = thisModule.counterI;
    }
    to
        input : AliasComponent!Input (
           id <- 'input_'+ct,
                 name <- nameE,
          type <- if typeEl.type = #string then 'string' else 'long' endif,
          cardinality <- #UNARY_REQUIRED)
    do{
        thisModule.counterI <- thisModule.counterI + 1;
        thisModule.inputP->including(input);
    }
}

lazy rule Element2Input{
    from
        el : WSDL!Element
    using{
        ct : Integer = thisModule.counterI;
        typeEl : WSDL!SimpleType = el.type;
    }
    to
        input : AliasComponent!Input (
          id <- 'input_'+ct,
          name <- el.Name,
```

```
31          type <- if typeEl.type = #string then 'string' else 'long' endif,
            cardinality <- if el.MaxOccurs = 'unbounded' then
                  if el.MinOccurs = '0' then #MULTIPLE_NOT_REQUIRED
                  else #MULTIPLE_REQUIRED
                  endif
36             else if el.MinOccurs = '0' then #UNARY_NOT_REQUIRED
                  else #UNARY_REQUIRED
                  endif
               endif)
    do{
41     thisModule.counterI <- thisModule.counterI + 1;
       thisModule.inputP->including(input);
    }
}
```

Listing 5.4 – Règle principale qui permet le déclenchement de la transformation

Afin d'illustrer, on reprend le WSDL du *Web Service* de l'application *Business*. La figure 5.1 correspond à un extrait du WSDL. A partir de ce modèle, on obtient par transformation, le modèle présenté en figure 5.2 qui est conforme au méta-modèle `AliasComponent` et qui correspond à l'abstraction du *Web Service*.

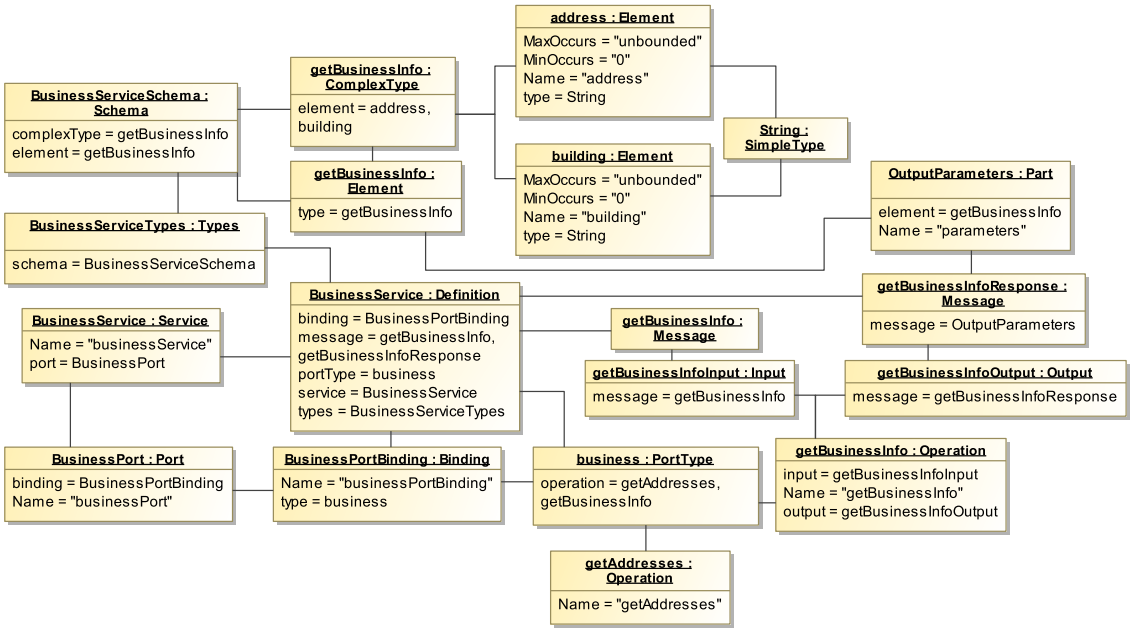


FIGURE 5.1 – Extrait du WSDL du *Web Service BusinessService*

Dans le cas des approches à composants telles que WCOMP ou FRACTAL, il serait nécessaire d'étudier les interfaces que proposent ces composants. Dans le cas de FRACTAL, il suffit de prendre l'interface fournie du composant pour obtenir l'ensemble des méthodes qu'il propose. Dans le cas de WCOMP, il faudrait étudier l'ensemble des ports d'entrée qui correspondent aux méthodes proposées par le composant.

Abstraction de la partie présentation d'une application

L'abstraction de la partie présentation d'une application consiste à transformer une IHM concrète (ou finale) en une IHM abstraite qui correspond au composant d'IHM abstraite (*UIComponent*) du méta-modèle `AliasComponent`. Ce sont les éléments graphiques qui constituent l'IHM

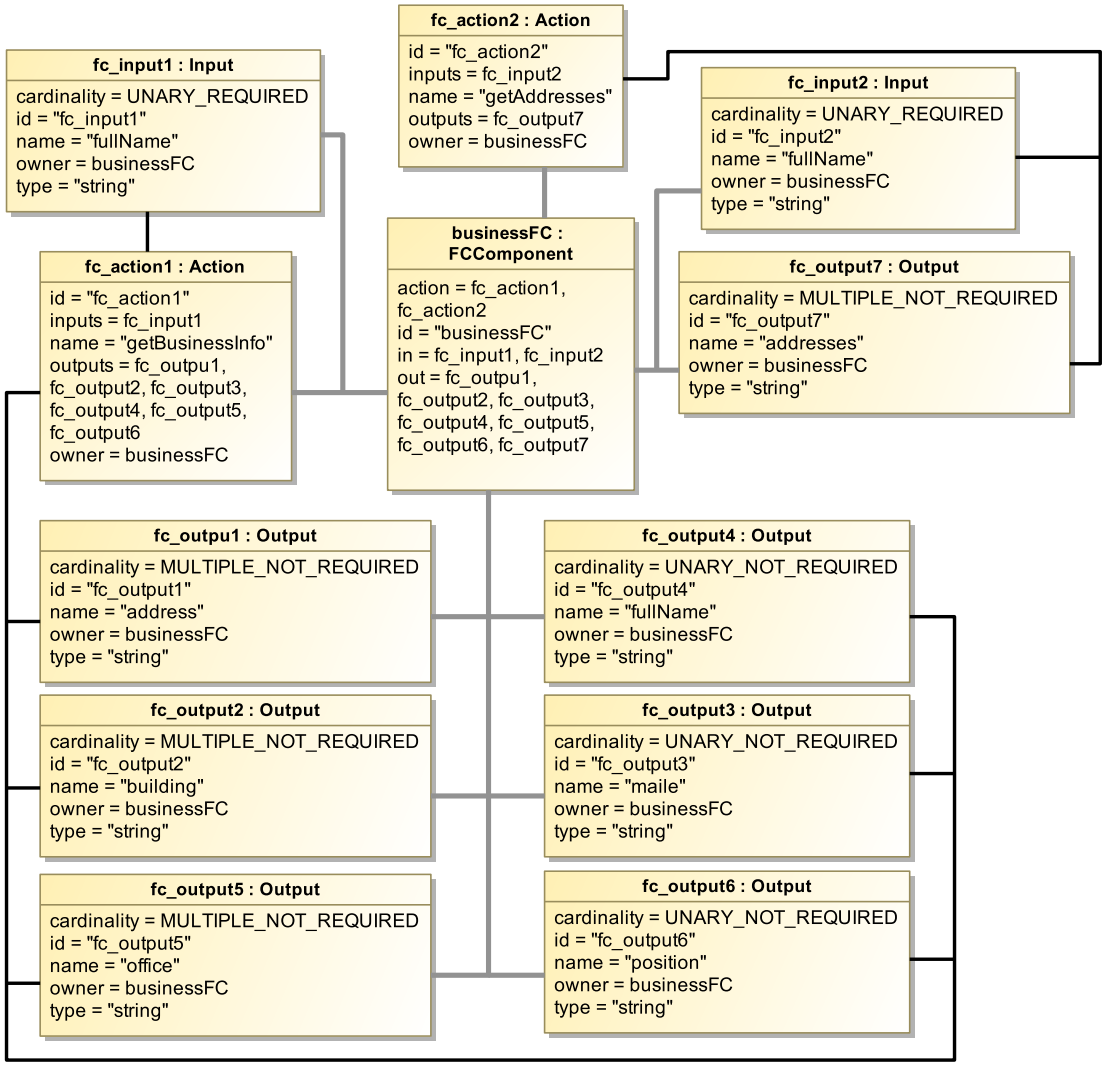


FIGURE 5.2 – Abstraction du service *BusinessService* au sein du méta-modèle `AliasComponent`

qui doivent devenir des ports d'entrée, de sortie et d'action. La formalisation d'une application (cf. section 4.2.1) spécifie que l'ensemble des ports d'un composant d'IHM doit être relié au composant fonctionnel ce qui signifie que seuls les éléments ayant un lien direct avec la partie fonctionnelle de l'application doivent être conservés. Mais il existe également un ensemble d'éléments qui ne sont pas liés.

Une IHM est constituée d'un ensemble d'éléments graphiques qui sont soit nécessaires pour utiliser la partie fonctionnelle, soit ils sont présents pour aider l'utilisateur, soit ils structurent les éléments graphiques. Les éléments qui sont présents pour aider l'utilisateur peuvent être intéressant à conserver car ils peuvent désigner un autre élément graphique qui lui est relié au fonctionnel. C'est le cas par exemple d'un label qui désigne un champ de texte. En utilisant les méta-modèles présentés dans [PDJR+08], il est alors possible de conserver plus d'informations durant l'abstraction des IHM et par la suite d'obtenir une meilleure concrétisation.

Enfin les éléments qui sont liés à la partie fonctionnelle doivent être créés en tant qu'entrée

s'ils sont reliés à une entrée ou à une sortie s'ils sont reliés à une sortie.

Le mécanisme d'abstraction mis en œuvre s'appuie sur l'utilisation d'une table d'abstraction qui fait correspondre à un *widget* concret sa représentation abstraite. La réalisation de cette abstraction, présentée ci-après, s'appuie sur des descriptions réalisées à partir de `Flex` (MXML et `ActionScript`) et sur des IHM de type "formulaire", dans le cas d'IHM plus complexes, il serait nécessaire de demander au développeur de réaliser cette abstraction manuellement. L'avantage de cette description est d'avoir un bon découpage entre la partie description des IHM et la partie description des interactions.

Ce qui suit présente ce qu'il est possible d'abstraire en utilisant une table d'abstraction et s'appuie sur le listing 5.5 qui correspond à l'IHM pour l'opération *getBusinessInfo* du service *BusinessService*.

```
1   <s:Button x="220" y="13" label="getBusinessInformation" id="button"
        click="button_clickHandler(event)"/>
    <s:Label x="10" y="21.5" text="FullName"/>
    <s:TextInput x="73" y="11.5" id="FullNameInput"/>
    <mx:HRule x="10" y="41" width="624"/>
6   <s:Label x="341" y="197" text="Address:"/>
    <s:List x="399" y="149" id="addressOutput"></s:List>
    <s:Label x="10" y="51" text="FullName: "/>
    <s:Label x="80" y="51" text="-" id="FullNameOutput"/>
    <s:Label x="10" y="80" text="Email:"/>
11  <s:Label x="10" y="116" text="Position:"/>
    <s:Label x="11" y="198" text="Office:"/>
    <s:Label x="165" y="198" text="Building:"/>
    <s:List x="48" y="150"  id="officeOutput"></s:List>
    <s:List x="222" y="148" id="buildingOutput"></s:List>
16  <s:Label x="80" y="80" text="-" id="EmailOutput"/>
    <s:Label x="80" y="116" text="-" id="PositionOutput"/>
</s:Application>
```

Listing 5.5 – Description de l'IHM pour l'opération *getBusinessInfo*

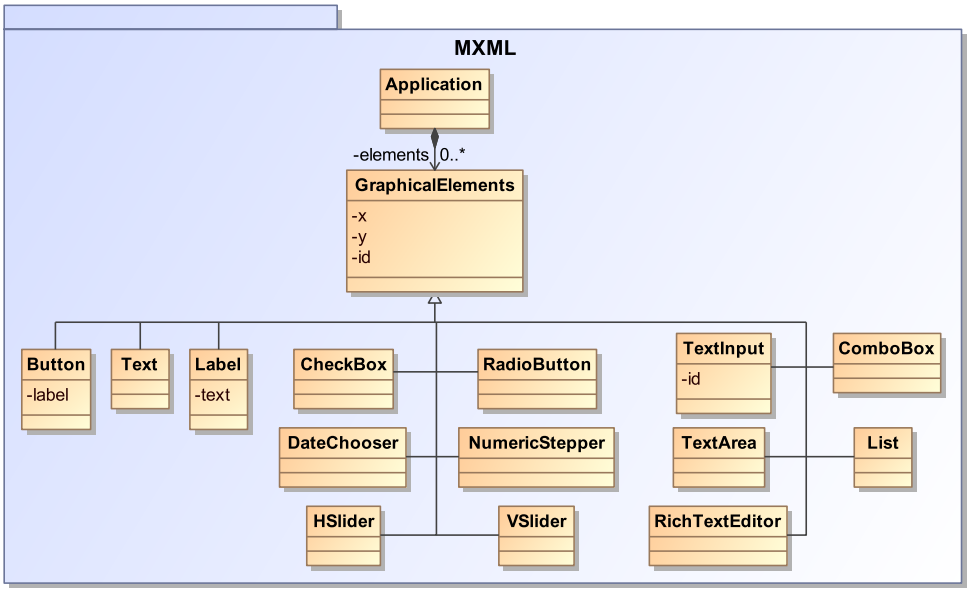


FIGURE 5.3 – Méta-modèle de MXML

Le méta-modèle simplifié de MXML est présenté en figure 5.3 et l'instance représentant le listing 5.5 est présentée en figure 5.4.

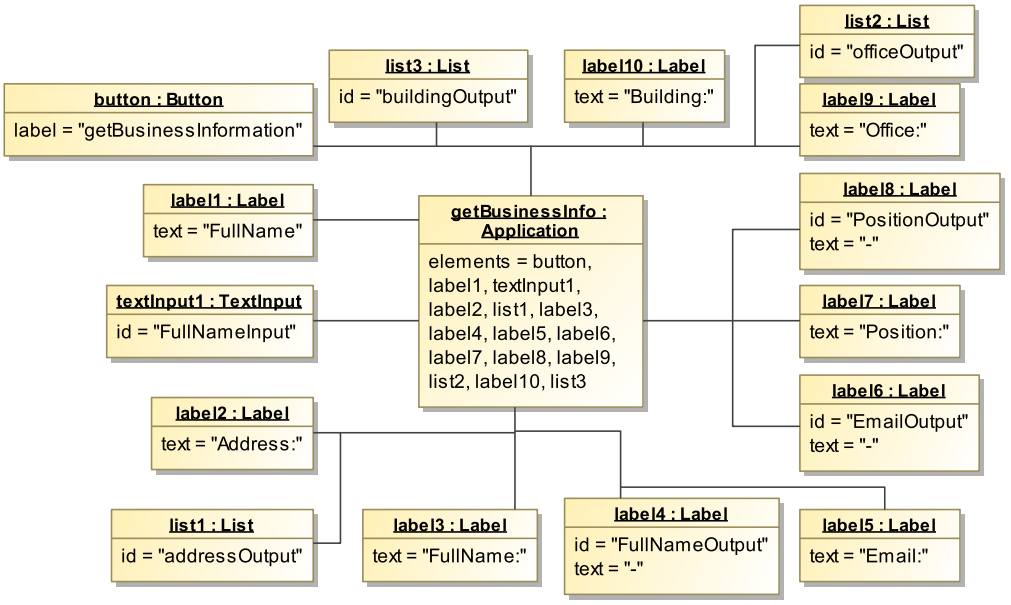


FIGURE 5.4 – Méta-modèle de MXML

L'utilisation d'une table d'abstraction permet d'associer à tout objet graphique une représentation sous forme d'entrée, sortie ou action avec des attributs différents selon l'objet graphique abstrait. L'avantage de cette approche est la facilité de mise en œuvre qui fait qu'il devient très facile d'abstraire des éléments d'IHM. L'inconvénient majeur de cette approche est le nombre d'éléments abstraits qui ne sont pas nécessaires ainsi que les erreurs qui peuvent apparaître dû au fait que l'on fait correspondre un élément concret à un élément abstrait sans considérer l'utilisation qui est fait de l'élément concret. C'est ainsi qu'il est possible d'abstraire un élément en tant qu'entrée alors que c'est une sortie.

La table 5.1 présente la mise en place d'une table d'abstraction pour permettre de donner une équivalence entre un élément concret et un élément abstrait. Cette table est basée sur les éléments présents au sein des IHM décrites en MXML.

L'utilisation de cette table d'abstraction permet d'obtenir l'abstraction suivante de la description de l'IHM (cf. listing 5.5) : l'ensemble des *Label* est transformé en $output$ sous la forme : $id(output_X) = ui\_outputX, name(output_X) = id$, $type(output_X) = string, cardinality(output_X) = 1$, on utilise l'identifiant du composant comme nom de sa représentation abstraite. Les *List* sont transformées en $input$ sous la forme : $id(input_Y) = ui\_inputY$, $name(input_Y) = id$, $type(input_Y) = string$, $cardinality(input_Y) = n$ et $selection(input_Y) = single$ où de la même manière que pour les éléments précédemment créés, on utilise l'identifiant comme nom. Le *TextInput* est abstrait en $input$ sous la forme : $id(input_Y) = ui\_inputY$, $name(input_Y) = id$, $type(input_Y) = string$ et $cardinality(input_Y) = 1$. Et pour finir le bouton est abstrait en *event* sous la forme : $id(eventt_Z) = ui\_eventZ$, $name(eventt_Z) = label$, $inputs(eventt_Z) = \emptyset$ et $outputs(eventt_Z) = \emptyset$. Pour le bouton, il est possible d'utiliser son label comme nom pour compléter la création de l'élément avec la bonne donnée à l'intérieur. Par contre, il n'est pas possible de pouvoir à chaque événement ses entrées et ses sorties.

Si la mise en œuvre d'une telle table d'abstraction et la transformation associée permet d'obtenir rapidement un premier résultat d'abstraction, il n'en reste pas moins que le résultat obtenu (cf. figure 5.5) ne correspond pas à ce que l'on attend car : (i) des éléments graphiques ont mal été abstraits, (ii) tous les *Labels* ont été abstraits comme des sorties alors que certains ne sont là que pour dénommer un élément et (iii) il manque les associations entre les événements et leurs

entrées et sorties. Pour permettre de palier à ce problème, il est nécessaire que le développeur modifie lui-même l'abstraction obtenue ou que des mécanismes d'abstraction plus complexe soient mis en place (cf. 6.2).

Elément Concret	Type d'abstraction	Attributs		
		Multiplicité	Sélection	Type
Button	*EVENT*	-	-	-
CheckBox	*INPUT*	*n*	*Mulitple*	*String*
RadioButton	*INPUT*	*n*	*Single*	*String*
DateChooser	*INPUT*	1	-	*Date*
H/VSlider	*INPUT*	1	-	*Number*
NumericStepper	*INPUT*	1	-	*Number*
TextInput	*INPUT*	1	-	*String*
TextArea	*INPUT*	1	-	*String*
RichTextEditor	*INPUT*	1	-	*String*
List	*INPUT*	*n*	*None,Single, Multiple*	*String*
ComboBox	*INPUT*	*n*	*Single*	*String*
Label	*OUTPUT*	1	-	*All*
Text	*OUTPUT*	1	-	*All*

TABLE 5.1 – Table d'abstraction d'éléments graphiques MXML

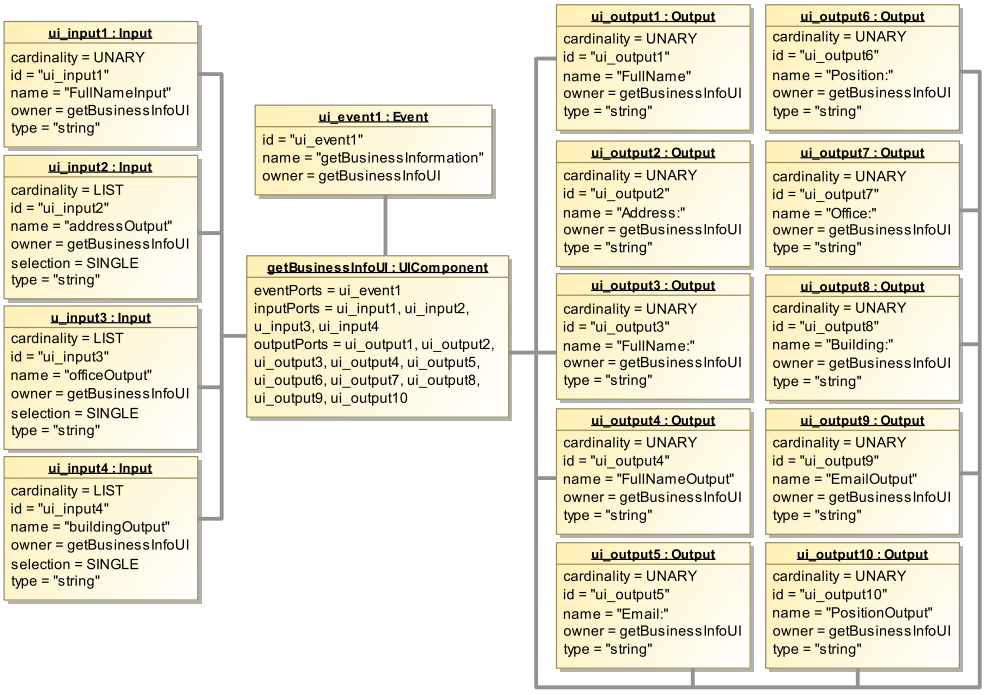


FIGURE 5.5 – Résultat de l'abstraction en utilisant la table d'abstraction

Abstraction de la partie interaction d'une application

L'abstraction de l'interaction présente au sein d'une application peut se faire par l'analyse de la partie Contrôleur que l'on trouve dans le patron d'architecture MVC en utilisant des méthodes de *reverse engineering*. Dans le cas d'application orientée web où la partie cliente se trouve être décrite en `Flex` (comme précédemment on a pu le voir pour la partie IHM) la partie interaction se situe dans l'*Action Script*. L'objectif est de donc de retrouver les opérations précédemment abstraites qui correspondent aux actions de la partie fonctionnelles. Une fois celles-ci retrouvées, il faut ensuite déterminer quels sont les éléments d'IHM qui fournissent les paramètres d'entrée et ceux qui reçoivent le message de retour. Ce sont par ailleurs les éléments qui ont été abstraits dans l'étape précédente lors de l'analyse des liens avec la partie fonctionnelle. L'abstraction des liens peut être faite en même temps.

Les liens d'événements vont relier l'élément déclencheur au niveau de la partie IHM à une opération de la partie fonctionnelle, les liens de donnée relient quant à eux les éléments d'IHM abstraits comme aux entrées et sorties de la partie fonctionnelle.

Le listing 5.6 présente l'ensemble de l'interaction présente au sein de l'IHM et qui permet de faire le lien entre le partie IHM et la partie fonctionnelle. Ici, on peut voir que le bouton qui possède l'identifiant *button* est l'élément déclencheur de l'appel de l'opération *getBusinessInfo*. On remarque également comment sont affectées les différentes valeurs de retours aux éléments graphiques présents dans l'IHM. Ainsi, l'élément graphique qui permet de recevoir le nom complet (*FullNameOutput*) doit être lié à la sortie *fullName* de l'opération.

```
      protected function button_clickHandler(event:MouseEvent):void
      {
        getBusinessInfoResult.token = businessService.getBusinessInfo(FullNameInput.text);
      }

      protected function Business_result(evt:ResultEvent):void
      {
        FullNameOutput.text = getBusinessInfoResult.lastResult.fullName;
        EmailOutput.text =  getBusinessInfoResult.lastResult.email;
        PositionOutput.text = getBusinessInfoResult.lastResult.position;
        for (var i:uint = 0 ; i < getBusinessInfoResult.lastResult.address.length; i++){
          addressList.addItem(getBusinessInfoResult.lastResult.address[i]);
        }
        for (var j:uint = 0 ; j < getBusinessInfoResult.lastResult.building.length; j++){
          buildingList.addItem(getBusinessInfoResult.lastResult.building[j]);
        }
        for (var k:uint = 0 ; k < getBusinessInfoResult.lastResult.office.length; k++){
          officeList.addItem(getBusinessInfoResult.lastResult.office[k]);
        }
      }
    ]]>
  </fx:Script>

  <s:Button x="220" y="13" label="getBusinessInformation" id="button"
     click="button_clickHandler(event)"/>
```

Listing 5.6 – Liens d'interaction entre la partie IHM et la partie fonctionnelle

La figure 5.6 illustre le résultat obtenu suite à l'analyse de l'interaction entre la partie fonctionnelle et la partie IHM (en partant d'une abstraction de l'IHM correcte). Ceci conclut l'abstraction d'une application. Ce qui suit présente l'abstraction de la composition fonctionnelle.

Abstraction de la composition fonctionnelle

L'abstraction de la composition fonctionnelle consiste en la création du composant composite (*Composite*) du méta-modèle `AliasComponent`. Ce composant possède les mêmes caractéristiques qu'un composant fonctionnel, mais il possède en plus un ensemble de sous-composants

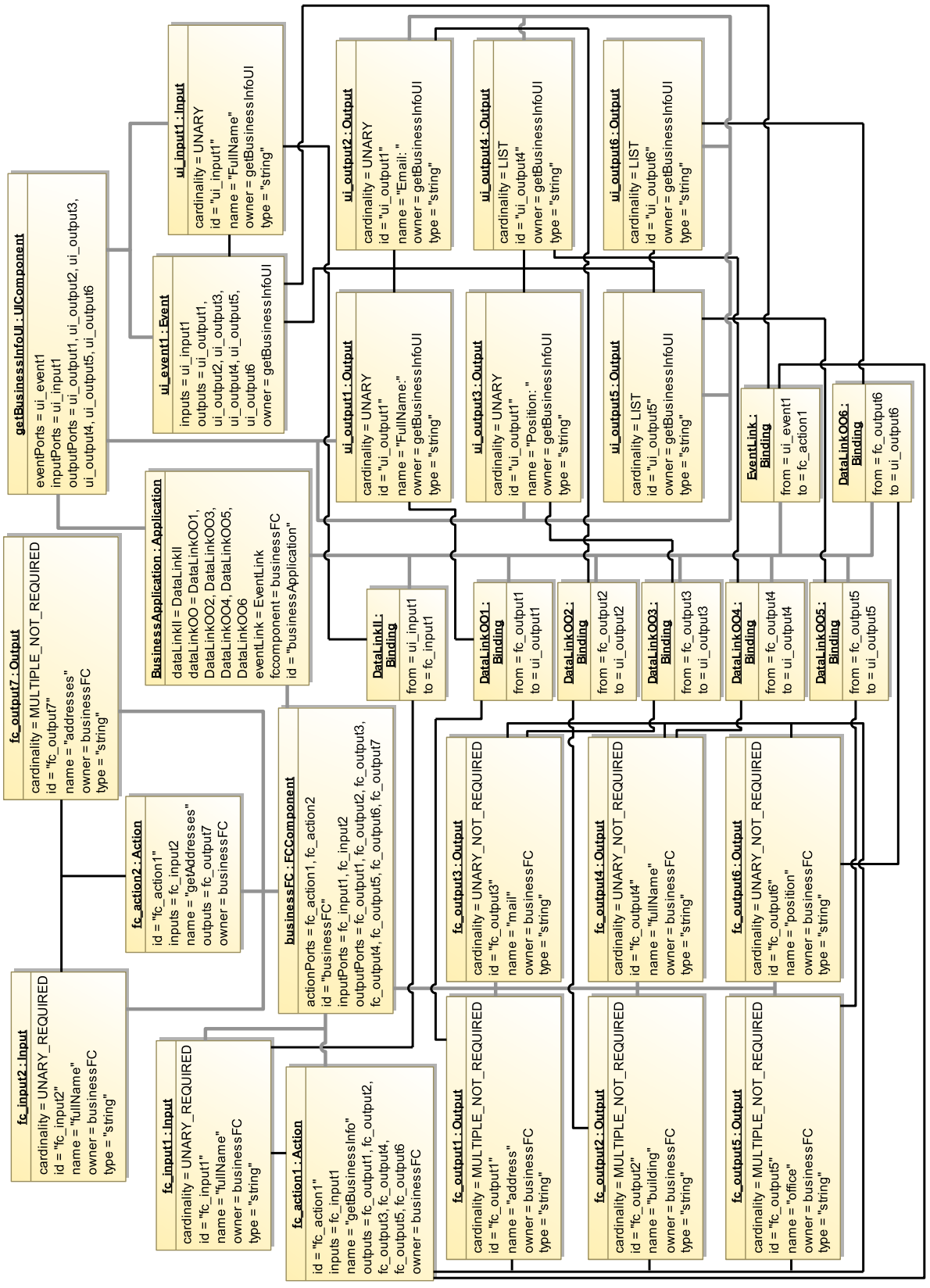


FIGURE 5.6 – Résultat de l'abstraction de l'application `Business` au sein d'`AliasComponent`

fonctionnels (qui correspondent aux composants précédemment créés par abstraction) et un ensemble de liens qui permet de relier ses ports (d'entrée, de sortie et d'action) aux ports de ses sous-composants mais également des liens qui relient les sous-composants entre eux.

Les approches à composants décrivent des assemblages ainsi que des composants composites. Le problème est que les liens relient des interfaces requises avec des interfaces fournies sans spécifier quel usage est fait des données. Il est donc nécessaire d'introspecter le code des composants pour savoir comment sont reliées les entrées et les sorties. De la même manière, il est possible de déterminer comme sont exécutées les opérations en instropectant le code des composants. Seule les approches telles que WCOMP ou les JAVABEANSdécrivent au travers des liens le flot de contrôle et donc l'enchaînement des méthodes à exécuter. De plus, dans WCOMP, il est également possible de déterminer les liens de données grâce aux rétro-appels faits sur le composant émetteur de l'événement.

Dans les approches à services, il est nécessaire d'analyser l'orchestration de services qui décrit le *workflow*. Cela comprend à la fois le flot de données et le flot de contrôle. Une orchestration qui est un service dans le modèle BPEL s'appuie sur deux ou trois éléments (le troisième élément étant optionnel comme pour la description des *Web Services*) : la description du *workflow*, la description du service qui va être présenté et optionnellement la description du schéma de données du service.

La première étape d'abstraction de l'orchestration de *Web Services* consiste à obtenir le composant composite avec l'ensemble de ses sous-composants. Pour obtenir le composant composite, on applique la même méthode que celle présentée pour l'abstraction des *Web Services* puisque l'on a la même description (WSDL et schéma de données). Pour obtenir les sous-composants et l'ensemble des liens de données et d'événements présents au sein de la composition, il est nécessaire d'analyser la description BPEL. Une représentation simplifiée du méta-modèle BPEL est donnée par la figure 5.7. Deux autres étapes sont nécessaires pour réaliser l'abstraction de l'orchestration de *Web Service* qui sont l'analyse du flot de contrôle et l'analyse du flot de données.

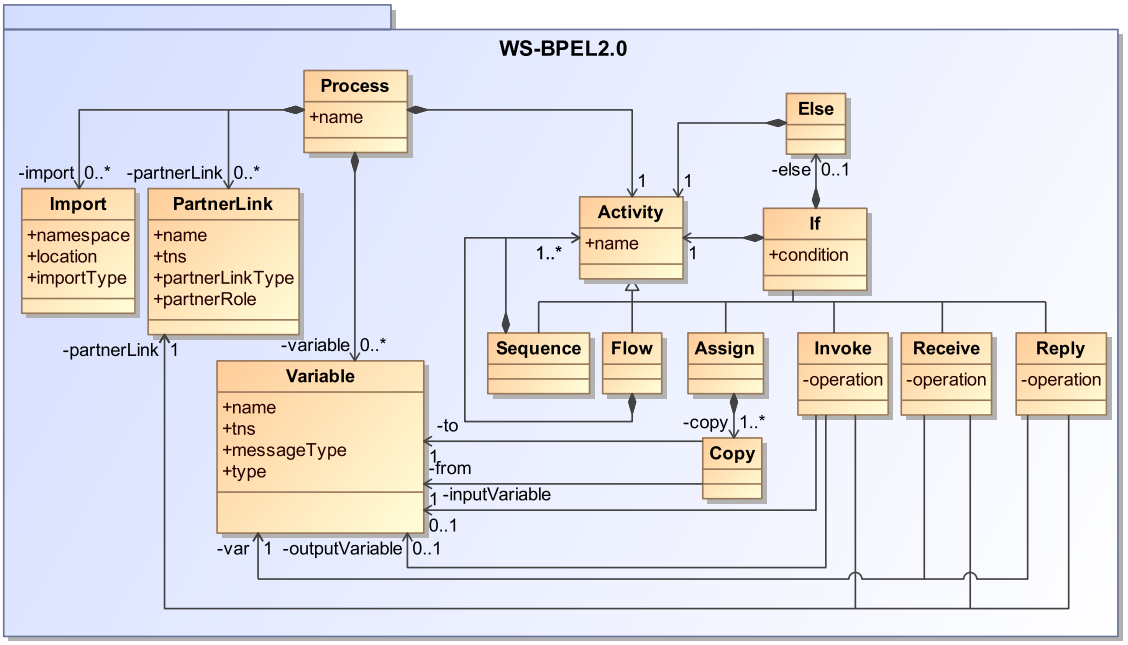


FIGURE 5.7 – Méta-modèle de BPEL

Analyse du flot de contrôle. Le flot de contrôle est défini dans l'ensemble des éléments présents dans le BPEL en-dehors des *Assign*. Les éléments indispensables pour une orchestration sont : le *receive* et le *reply*. Le *receive* correspond à la réception du message associé à une opération et permet d'affecter le contenu du message à une variable d'entrée (ce qui sert pour la partie création du flot de données). L'activité *reply* permet d'envoyer le message de sortie qui correspond à la valeur de retour de l'opération qui a été appelée. Ces deux activités correspondent aux points d'entrée et de sortie du *workflow*. Ces activités sont contenues dans une activité *sequence* qui permet de séquencer les actions qui vont être faites.

Les activités qui nous intéressent ensuite sont les activités de : *flow*, *sequence* et *invoke*. L'activité *flow* exprime une exécution en parallèle, cette activité peut imbriquer des activités de *flow* ou de *sequence*. L'activité *sequence* exprime une exécution séquentielle, une activité doit être finie pour passer à la suivante. Enfin l'activité *invoke* exprime l'appel à une opération d'un des services orchestrés. Ces activités vont permettre de savoir comment doivent être créés les liens d'événement entre les *actions* du composant composite et des sous-composants.

Dans le cas d'un *flow*, il faut que l'ensemble des opérations présentes dans le *flow* soit exécuté en même temps. Pour ce faire, l'*action* est reliée à l'ensemble des *actions* présentes dans le *flow*. C'est l'action de l'activité précédente (*Invoke* ou *Receive*) qui doit être reliée à l'ensemble des actions impliquées dans le *flow*.

Le *if...else* est quand à lui transformé en élément condition du méta-modèle `AliasComponent`. Et les liens sont créés en fonctions des activités présentes dans le *if* puis dans le *else*.

A la fin de cette étape, l'ensemble des *ActionLink* présents dans le méta-modèle `AliasComponent` ont été créés et reliés aux *actions* correspondantes.

Analyse du flot de données. Le flot de données est représenté au sein des activités *assign*. Ces activités permettent de faire le lien entre les éléments des variables utilisées au sein de l'orchestration qui correspondent aux messages d'entrées et de sorties des différentes opérations. C'est par l'étude des *assign* qu'il est ainsi possible de déterminer comment doivent se faire les liens de données. Mais pour être sûr de faire les liens avec les bons éléments précédemment abstraits, il est nécessaire d'identifier à quels éléments de quel opération cela doit être fait. Pour ce faire, il est nécessaire de regarder dans un premier temps les variables qui ont été définies. Les variables sont définies pour un message donné et permettent les échanges avec les opérations. Il est ainsi possible de déterminer quel est le format de données qu'il y a derrière une variable car on sait à quel service elle est associée et quel est son type. Les *invokes* permettent quant à eux d'associer chaque variable à une opération (afin de savoir par la suite comment doivent s'effectuer les liens au niveau de la description de la composition fonctionnelle dans `AliasComponent`) et également de savoir si c'est une variable d'entrée ou de sortie.

Dans le listing 5.7, le port de destination reçoit une information de deux sources. Ici un calcul est fait pour permettre la concaténation des deux éléments sources. L'abstraction est alors représentée par deux liens qui proviennent des deux sources et qui sont connectés à la même destination. Deux sorties pourraient être également reliées à une seule et même sortie afin de fusionner l'ensemble des valeurs. Ce qui correspond à des plusieurs liens de sorties reliés à une même sortie. Enfin une entrée peut-être liée à plusieurs entrées, dans ce cas là, on a un partage de données. Ces informations doivent être conservées pour permettre de savoir quel est l'usage qui est fait des données et pour permettre de savoir ce que l'on doit conserver au niveau de l'IHM. Le résultat obtenu à la fin de l'abstraction correspond à la figure 4.15.

```
<assign name="AssignToCallBusiness">
  <copy>
    <from>concat($GetByCardOut.parameters/return/firstName, ' ',
                 $GetByCardOut.parameters/return/lastName)
    </from>
    <to>$GetBusinessInfoIn.parameters/arg0</to>
  </copy>
</assign>
```

Listing 5.7 – Concaténation de deux variables pour fournir une entrée à la seconde opération

Conclusion

Les abstractions présentées permettent d'obtenir une description des applications et de la composition fonctionnelle au sein du méta-modèle `AliasComponent`. L'abstraction du noyau fonctionnel et de la composition fonctionnelle s'appuie sur l'utilisation des méta-modèles correspondants et de règles de transformations. Ces transformations assurent que le résultat obtenu est bien conforme au méta-modèle `AliasComponent`. L'abstraction de l'IHM nécessite plus de travail pour permettre d'identifier, pour chacun des éléments associés au noyau fonctionnel, son type (entrée ou sortie) ainsi que le label associé. L'identification peut se faire par l'étude des interactions avec le noyau fonctionnel et la recherche des labels par l'étude du placement des éléments les uns par rapport aux autres. La seule table d'abstraction mise en place ne suffit pas pour obtenir une abstraction qui reflète l'IHM existante. Enfin, l'abstraction des liens d'interaction entre l'IHM et le noyau fonctionnel correspond à un travail de compilation afin de pouvoir identifier dans le code les éléments et leurs associations.

Le fait d'abstraire des IHM existantes donne également la possibilité de conserver les informations présentes dans celles-ci en vue de la concrétisation du résultat. Ces informations permettraient de compléter les méta-modèles présentés dans [PDJR+08] en vue d'obtenir un meilleur rendu final. Ces informations supplémentaires permettraient ainsi d'obtenir un meilleur rendu et éviteraient d'avoir à mettre en place une table de correspondance entre éléments abstraits et éléments concrets comme présentée ci-après (cf. section 5.1.2).

5.1.2 Concrétisation du résultat obtenu pour permettre la validation

La concrétisation permet d'obtenir une IHM concrète à partir de la représentation abstraite présente dans le composant d'IHM. Cette concrétisation permet ainsi d'avoir une vision globale du résultat obtenu et permet ainsi au développeur de vérifier que celui-ci est correct.

La concrétisation s'appuie sur un des principes de l'abstraction d'IHM qui est la mise en place d'une table de concrétisation. En effet, de la même manière, pour une abstraction donnée correspond une concrétisation. Celle-ci peut-être étendue à plusieurs concrétisations possibles qui demanderaient au développeur de faire un choix pour avoir l'élément voulu. La table 5.2 représente la table de concrétisation qui a été mise en place pour permettre la concrétisation des éléments d'IHM en éléments graphiques `Java` Swing. On retrouve ainsi l'ensemble des attributs qui caractérisent une entrée, une sortie ou un événement. On peut remarquer que l'on a deux colonnes pour la concrétisation qui sont : Concrétisation et Autres Concrétisations. La première permet de fournir un rendu unique à chaque élément que l'on peut décrire au sein d'`AliasComponent` pour une IHM. La seconde colonne permet de fournir des alternatives possibles au développeur. Pour ce faire, cela nécessite la mise en place d'un outil qui permet d'interroger le développeur lorsque plusieurs concrétisations sont possibles. Tous les éléments décrits au sein d'`AliasComponent` ne possèdent pas plusieurs concrétisations. Ainsi, si on considère une entrée de type *String* qui est de cardinalité *Unary*, la concrétisation immédiate est un *JTextField* dans le cas de `Java` Swing, une alternative possible est de proposer un *JTextArea* qui permet de fournir du texte plus long et d'avoir un plus grand espace de saisie.

De la même manière, il est possible de décrire une table de concrétisations pour obtenir un rendu `Flex` ou XAML. Il est ainsi possible de facilement obtenir plusieurs rendus dans des langages cibles différents.

Par ailleurs, durant la phase d'abstraction, plus d'informations concernant les éléments abstraits pourraient être conservées afin de permettre une meilleure concrétisation par la suite. Cela permettrait ainsi d'éviter de devoir demander au développeur de faire un choix parmi les différentes concrétisations possibles. Celles-ci seraient contraintes par les éléments existants qui ont été abstraits.

Pour réaliser ces transformations de concrétisation et ainsi obtenir le code des IHM, il est possible d'utiliser des outils de génération de code comme `Acceleo`. Cet outil s'appuie sur l'utilisation

d'un méta-modèle (ici, le méta-modèle `AliasComponent`) et un patron de génération de code qui va utiliser un modèle `AliasComponent` pour générer le code associé à cette instance.

Element	uitype	cardinality	selection	Concrétisation	Autres Concrétisations
INPUT	*Number*	*Unary*	-	*JTextField*	*JSlider* ou *JSpinner*
		List	*None*	*JList*	-
			Single	*JList*	*JRadioButton*
			Multiple	*JList*	*JCheckBox*
	String	*Unary*	-	*JTextField*	*JTextArea*
		List	*None*	*JList*	-
			Single	*JList*	*JComboBox* ou *JRadioButton*
			Multiple	*JList*	*JCheckBox*
	Date	*Unary*	-	*JTextField*	*JXDatePicker* (nécessite SwingX)
		List	*None*	*JList*	-
			Single	*JList*	*JRadioButton*
			Multiple	*JList*	*JCheckBox*
	Boolean	*Unary*	-	*JCheckBox*	*JTextField*
		List	*None*	*JList*	
			Single	*JRadioButton*	-
			Multiple	*JCheckbox*	*JTable*
OUTPUT	*Number*	*Unary*	-	*JLabel*	-
		List	-	*JList*	-
	String	*Unary*	-	*JLabel*	-
		List	-	*JTextArea*	*JList*
	Date	*Unary*	-	*JLabel*	-
		List	-	*JList*	-
	Boolean	*Unary*	-	*JLabel*	*Icon*
		List	-	*JList*	-
EVENT	-	-	-	*JButton*	*JMenuItem*

TABLE 5.2 – Table de concrétisations en `Java` Swing

5.1.3 Conclusion

Les transformations mise en œuvre pour permettre l'abstraction et la concrétisation s'appuient sur l'utilisation de l'ingénierie dirigée par les modèles. Ces transformations permettent ainsi d'obtenir à partir d'applications existantes une représentation au sein du méta-modèle `AliasComponent`. De même le résultat obtenu suite à la réalisation de la composition peut être concrétisé en générant du code afin de visualiser le résultat. Les transformations d'abstraction des IHM nécessitent d'être enrichies afin d'améliorer le résultat obtenu et d'ainsi éviter une charge supplémentaire au développeur afin de corriger l'abstraction qui est faite.

L'illustration des abstractions s'est fait sur une technologie en particulier pour chacun des cas (que ce soit pour l'application ou pour la la composition fonctionnelle). Pour obtenir d'autres abstractions, il est nécessaire de réaliser autant de transformations que de technologies que l'on souhaite abstraire en vue de les composer. De même pour la concrétisation, il est nécessaire de décrire autant de transformations que de technologies visées.

5.2 Atelier d'aide à la composition d'IHM

Afin de valider les algorithmes de composition ainsi que la démarche mise en œuvre, un atelier d'aide à la composition d'IHM a été développé (cf. section 5.2.1). Cet atelier s'appuie sur une description abstraite des applications et de la composition fonctionnelle au sein du méta-modèle `AliasComponent`. L'usage de l'atelier ainsi que le processus mis en œuvre pour la réalisation de la composition d'IHM ont été validés par des tests utilisateurs (cf. section 5.2.2).

5.2.1 Présentation de l'atelier

L'atelier se compose de deux parties : (i) un *front-end* pour l'interaction et (ii) un *back-end* pour la composition. Le *front-end*, réalisé en `Java`, permet au développeur d'intéragir avec le moteur de composition. Il fournit une IHM qui regroupe l'ensemble des éléments nécessaires à la réalisation de la composition. Le *back-end*, décrit en `Prolog`, réalise la composition à partir des éléments fournis par le développeur au travers du *front-end*.

Front-end `Java` pour l'interaction avec le développeur

Cette partie de l'application fournit une IHM permettant au développeur de réaliser la composition sans se préoccuper du moteur de composition qui a été développé en `Prolog`. Cette IHM permet :

- de charger un ensemble d'applications déjà abstraites qui sont impliquées dans la composition fonctionnelle ;
- de charger une ou plusieurs compositions fonctionnelles abstraites ;
- de réaliser la composition ;
- de visualiser l'assemblage de composants qui correspond soit à une application soit à une composition fonctionnelle ;
- d'obtenir un rendu graphique des IHM.

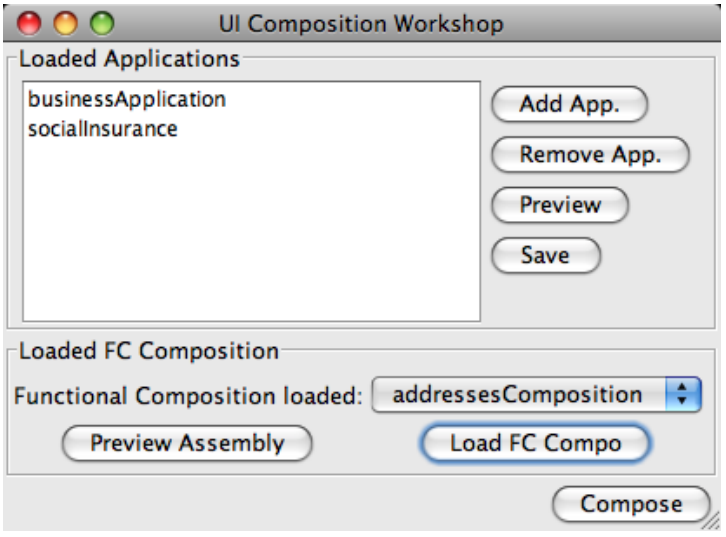


FIGURE 5.8 – IHM principale de l'atelier

La figure 5.8 correspond à l'écran de départ sur lequel arrive le développeur. Il permet d'atteindre toutes les fonctionnalités (chargement, *preview*, lancement de composition). On utilise des

fichiers de description des applications et de la composition fonctionnelle conforme au méta-modèle `AliasComponent` afin de les charger au sein de l'atelier. Ces fichiers de descriptions correspondent aux abstractions qui ont été faites au préalable. Suite au chargement de l'ensemble des applications et d'au moins une composition fonctionnelle, le développeur peut réaliser la composition d'IHM.

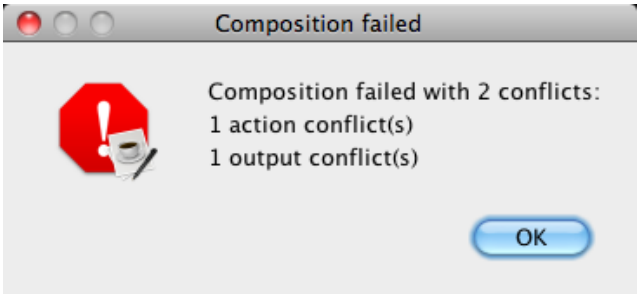


FIGURE 5.9 – Remontée des conflits

Suite au déclenchement de la composition d'IHM, l'ensemble des données sont transmises au moteur de composition (*back-end*). Celui-ci applique l'ensemble des définitions au chapitre 4 en section 4.2.3. Suite à cela, si le moteur de composition détecte des conflits, ceux-ci sont remontés aux développeurs. Une fenêtre (cf. figure 5.9) lui indique combien de conflits ont été détectés et sur quel type d'éléments.

Ensuite, des fenêtres de résolution de conflits sont présentées au développeur. La figure 5.10 illustre une fenêtre de résolution de conflits associés à une *ACTION* alors que la figure 5.11 présente une fenêtre de résolution de conflits pour une sortie. On retrouve au sein de ces deux fe-

FIGURE 5.10 – Résolution d'un conflit associé à une action

nêtres les quatre politiques de résolutions de conflits qui ont été présentés en section 4.2.3. Ainsi,

FIGURE 5.11 – Résolution d'un conflit associé à une sortie

le développeur peut soit réutiliser des éléments existants en sélectionnant un ou tous les éléments, soit créer un nouvel élément en réutilisant les informations présentes dans les éléments en conflits ou fournissant l'ensemble des informations qui caractérisent l'élément.

Suite à la résolution, le processus de composition est achevé par la création de la nouvelle application. Cette application peut-être visualisée sous la forme d'un assemblage de composants comme illustré dans la figure 5.12.

Enfin, le développeur peut obtenir un rendu graphique en `Java` Swing de l'IHM afin de se rendre compte du résultat obtenu par le moteur de composition. La figure 5.13 illustre l'IHM que l'on a pour la composition fonctionnelle permettant l'obtention de l'ensemble des adresses provenant du *Web Service* d'assurance sociale et du *Web Service* d'information professionnelle (cf. section 2.3.3).

Back-end `Prolog` pour la réalisation de la composition

Le *back-end* en `Prolog` correspond au moteur de composition. Il implémente l'ensemble des propriétés et définitions qui se trouvent dans le chapitre 4. Il permet de réaliser la composition d'IHM et de détecter les conflits qui ont été définis en section 4.2.3. Il fournit en résultat une nouvelle application. Cette application contient l'ensemble des composants d'IHM créés par le moteur de composition ainsi que l'ensemble des liens qui relient les composants d'IHM créés avec le composant composite correspondant à la composition fonctionnelle.

Le détail d'implémentation du moteur de composition se trouve en annexe B. L'implémentation du moteur de composition a été réalisée en `Swi-Prolog` et l'interfaçage avec l'application en `Java` est faite au travers de l'API `jpl`.

5.2.2 Tests utilisateurs réalisés sur l'atelier

Pour valider l'approche, des tests utilisateurs ont été menés. Ces tests portent sur la détection et la résolution des conflits durant le processus de composition d'IHM dirigée par la com-

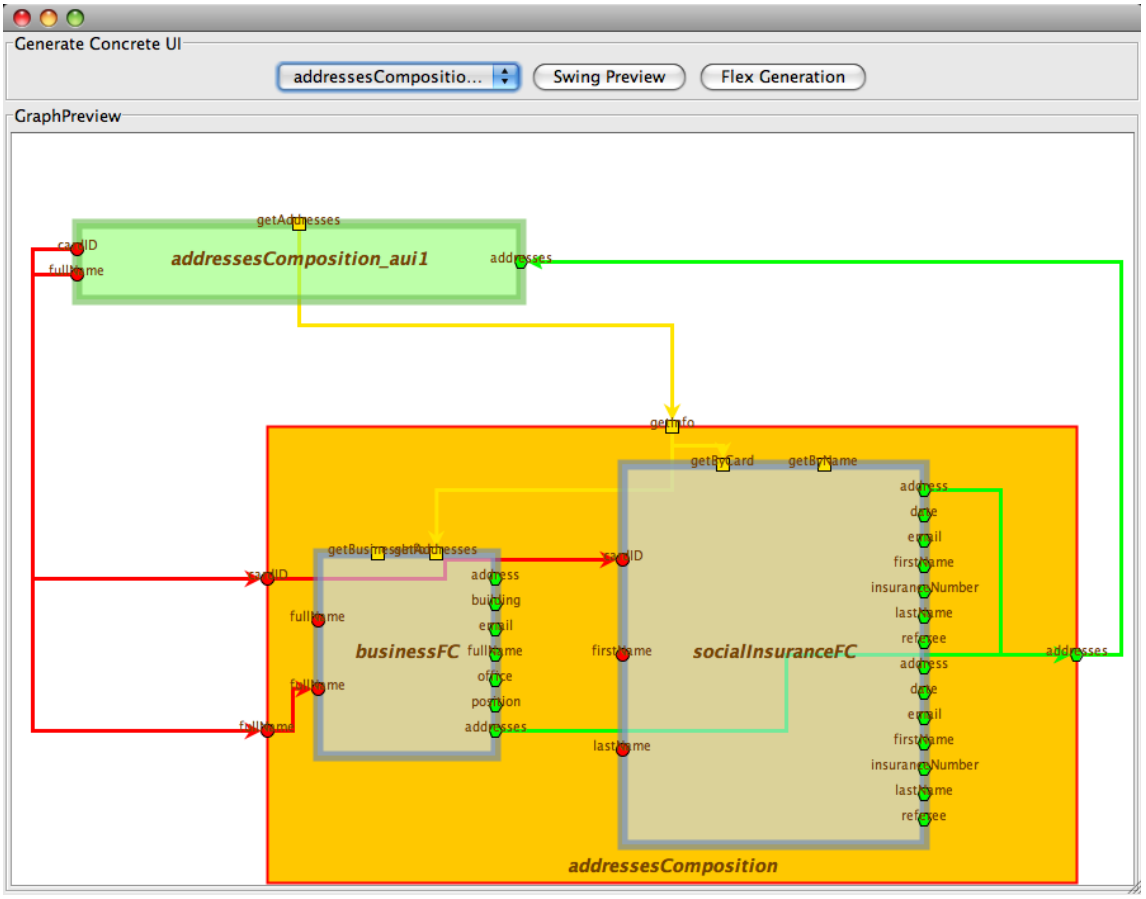


FIGURE 5.12 – Visualisation de l'assemblage de composants correspondant au résultat de la composition

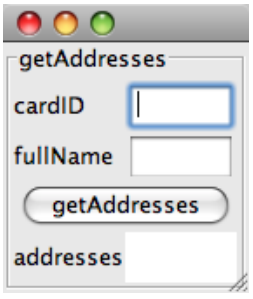


FIGURE 5.13 – Rendu de l'IHM en Java Swing

position fonctionnelle. Ainsi, l'objectif de ces évaluations est de vérifier que les conflits que l'on détecte grâce au moteur de composition, ainsi que les solutions que l'on propose pour résoudre ces conflits sont bien en adéquation avec les problèmes rencontrés par le développeur lors de la composition d'IHM ainsi qu'avec les solutions mises en œuvre pour les résoudre. Ainsi, cette évaluation permet de contrôler que l'aide que l'on fournit au développeur est utile, qu'elle cor-

respond à ses attentes et qu'elle y répond.

Ce qui suit présente : (i) le scénario mis en place pour réaliser l'évaluation, (ii) le protocole de l'évaluation et (iii) le questionnaire permettant à l'utilisateur de donner ses retours.

Scénario mis en place

Le scénario proposé pour réaliser l'évaluation dérive des exemples qui ont été présentés dans le chapitre 2. L'objectif est de réaliser, à partir des deux applications de base que sont l'application *Business* qui permet d'obtenir des informations professionelles sur une personne et l'application *SocialInsurance* qui permet d'obtenir des informations sur un assuré social, une application qui permet d'obtenir l'ensemble des adresses d'une personne (regroupées en un seul résultat) à partir de son numéro d'assuré et de son nom. Ce scénario correspond à la troisième composition fonctionnelle (cf. 2.3.3). La composition fonctionnelle ainsi qu'une possibilité de résultat sont présentées dans la figure 5.12. Afin d'ajouter d'autres potentiels conflits à gérer, les deux IHM sont quelque peu modifiées. Le libellé des deux adresses deviennent : PersonalAddress (pour l'application *SocialInsurance*) et ProfessionalAddress (pour l'application *Business*). Enfin, le type d'objet graphique utilisé pour afficher l'adresse obtenue dans l'application *SocialInsurance* est remplacé par une liste.

D'un point de vue purement IHM, ce scénario pose les problèmes suivants :

- quel bouton choisir pour permettre de déclencher l'appel de la nouvelle opération ? faut-il créer un nouveau bouton ou choisir un bouton existant ?
- comment doivent se fusionner les adresses afin que l'on ait qu'un objet graphique qui les regroupe toutes ? faut-il réutiliser un des éléments existants ? ou créer un nouvel élément ?

Protocole d'évaluation

Profil évaluateur. L'évaluation s'effectue auprès de développeurs experts dans la réalisation de compositions fonctionnelles et qui savent donc créer de nouveaux noyaux fonctionnels soit par orchestrations des Web Services ou par assemblage de composants. Ces développeurs possèdent également des connaissances le développement des IHM. Pour eux, le développement des IHM correspond à un besoin afin d'avoir un moyen d'interagir avec la composition fonctionnelle.

Matériels utilisés. Pour réaliser l'évaluation, on fournit à l'utilisateur des maquettes représentant les IHM associées à chacun des services pour les deux applications que l'on souhaite composer ainsi que la description des *Web Services* et les liens qu'ils ont avec l'IHM et pour finir la composition fonctionnelle sous la forme d'un flot de contrôle et de données. On fournit également des éléments vierges afin que l'utilisateur puisse créer ses propres objets graphiques dans le cas où aucun ne conviendrait. Une paire de ciseaux et un stylo ainsi qu'un espace vierge pour réaliser la composition des IHM.

Déroulement de l'évaluation. L'évaluation se déroule en trois étapes qui sont les suivantes : (i) réalisation de la composition d'IHM de manière manuelle avec verbalisation de la part de l'utilisateur, (ii) réalisation de la composition d'IHM au sein de l'atelier et (iii) évaluation de l'approche à l'aide d'un questionnaire.

1ère étape : explication et réalisation de la composition d'IHM. Cette explication est fournie par l'observateur. "Vous devez, à partir des IHM des applications à composer, fournir une nouvelle IHM qui permette d'obtenir la fusion de l'ensemble des adresses au sein d'un seul élément graphique. L'obtention de ces adresses se fait en ne fournissant que le numéro d'assuré social. Réalisez l'IHM correspondante en réutilisant les éléments à votre disposition. Dans le cas où vous rencontrez un problème ou un choix, expliquez quel est ce problème et comment vous comptez le

résoudre.". Durant cette étape, l'utilisateur doit verbaliser l'ensemble de sa démarche. Expliquer quels sont ses choix et pourquoi il les a faits.

2nd étape : réalisation de la composition d'IHM avec l'atelier. Lors de cette étape, il réalise la composition des applications et plus particulièrement des IHM au sein de l'atelier. L'observateur prépare l'atelier en chargeant les deux applications ainsi que la composition fonctionnelle. Le processus de composition est ensuite lancé. Les conflits apparaissent alors. Il est alors demandé à l'utilisateur de résoudre les conflits qui ont été détectés par l'atelier.

3ème étape : évaluation de l'approche. Lors de cette étape, l'utilisateur évalue la composition d'IHM qu'il a réalisée et plus particulièrement les conflits qu'il a du résoudre avec les possibilités de résolution de conflits et la composition d'IHM fournies par l'atelier. Pour chaque conflit, l'utilisateur doit spécifier s'il a rencontré ce conflit, si le choix qu'il a fait est bien présent dans les choix fournis. Pour chaque conflit sur les informations fournies sont compréhensibles pour appréhender sa résolution. Ensuite pour la résolution en elle-même, il doit spécifier si les choix sont clairs et compréhensibles. La mise en place d'un questionnaire permet à l'utilisateur d'évaluer les différents points précédemment cités (cf. table 5.3). Un débriefing est également réalisé pour permettre de clarifier des points d'incompréhension et de compléter les retours du questionnaire.

Résultats obtenus et conclusions

Les évaluations effectuées auprès de cinq développeurs d'applications interactives par composition fonctionnelle confirment que l'approche proposée est intéressante. Ils ont pu obtenir avec l'atelier un résultat comparable à celui réalisé avec les maquettes. Par contre, il ressort également que l'utilisation de l'atelier n'est pas facile et qu'il est nécessaire de fournir des explications pour comprendre les possibilités offertes pour résoudre les conflits.

La figure 5.14 reprend les résultats obtenus de la part de quatre des cinq utilisateurs. En effet, un des utilisateurs n'a pas jugé pertinent les questions concernant le conflit sur les boutons et n'y a donc pas répondu. Ce qui ressort de ces évaluations, c'est que les choix proposés sont :

- utiles et qu'ils permettent bien de résoudre les conflits et d'obtenir ce que les utilisateurs souhaitaient.
- compréhensibles dans l'ensemble mais que la clarté de ceux-ci pourraient être améliorés.
- trop nombreux car il y a beaucoup de choix possibles et la différence entre chaque n'est pas évidente (lié au fait que la compréhension pourrait être améliorée).

Un autre point qui ressort des questionnaires est que la notion de conflits n'est pas évidente car il peut y avoir différentes interprétations possibles et que le terme n'est pas forcément approprié. En effet, ce qui ressort, c'est plus une notion de choix que de conflit. Dans le cas où l'utilisateur a plusieurs possibilités qui lui sont offertes mais qui peuvent lui convenir, il ne considère pas que cela soit un conflit. De même, la réutilisation d'un élément qui doit être renommé (changer le label associé) n'est pas non plus un conflit (du point de l'utilisateur) alors que l'atelier le détecte comme tel. Il est donc nécessaire de faire un effort sur la définition que l'on a de conflit afin d'éviter par la suite tout problème de compréhension et d'appréhension de la part de l'utilisateur.

Les débriefings réalisés en fin d'évaluation ont permis d'éclaircir des points d'incompréhension des utilisateurs. Ils ont fait également ressortir trois points essentiels que sont : la modification de la détection des conflits, la mise en avant du point de conflit et la modification des choix de résolution de conflit.

Détection des conflits. En ce qui concerne la détection des conflits proposée par l'atelier, l'utilisateur 1 propose de pouvoir ajuster le degré de détection (de fin à large) afin de ne pas détecter deux éléments dont le nom diffère d'une majuscule comme étant en conflit. Il serait ainsi donc bon de paramétrer la détection pour alléger encore les résolutions que doit effectuer manuellement le

Général							
Intérêt de l'approche	inintéressant	1 ○	2 ○	3 ○	4 ○	5 ○	Très intéressant
Nombre de conflits trouvés par l'utilisateur :							
Nombre de conflits communs avec l'atelier :							

Conflits							
Conflits sur les adresses en sorties							
Avez-vous été confronté(e) à ce conflit ?	oui ○			○ non			
Si oui, la manière dont vous avez résolu le conflit faisait-elle partie des choix possibles ?							
Information sur le conflit							
Le conflit est-il compréhensible ?	oui ○			○ non			
Si non, que faudrait-il pour le rendre compréhensible ?							
Choix de résolution du conflit							
Les choix de résolution sont ils :							
	Insuffisants	1 ○	2 ○	3 ○	4 ○	5 ○	Trop nombreux
	Incompréhensibles	1 ○	2 ○	3 ○	4 ○	5 ○	Compréhensibles
	Inutiles	1 ○	2 ○	3 ○	4 ○	5 ○	Utiles
	Redondants	oui ○			○ non		
Suggestion :							
Conflits sur les boutons							
Avez-vous été confronté(e) à ce conflit ?	oui ○			○ non			
Si oui, la manière dont vous avez résolu le conflit faisait-elle partie des choix possibles ?							
Information sur le conflit							
Le conflit est-il compréhensible ?	oui ○			○ non			
Si non, que faudrait-il pour le rendre compréhensible ?							
Choix de résolution du conflit							
Les choix de résolution sont ils :							
	Insuffisants	1 ○	2 ○	3 ○	4 ○	5 ○	Trop nombreux
	Incompréhensibles	1 ○	2 ○	3 ○	4 ○	5 ○	Compréhensibles
	Inutiles	1 ○	2 ○	3 ○	4 ○	5 ○	Utiles
	Redondants	oui ○			○ non		
Suggestion :							

Intérêt de l'approche	Inintéressant	1 ○	2 ○	3 ○	4 ○	5 ○	Très intéressant

TABLE 5.3 – Grille d'évaluation

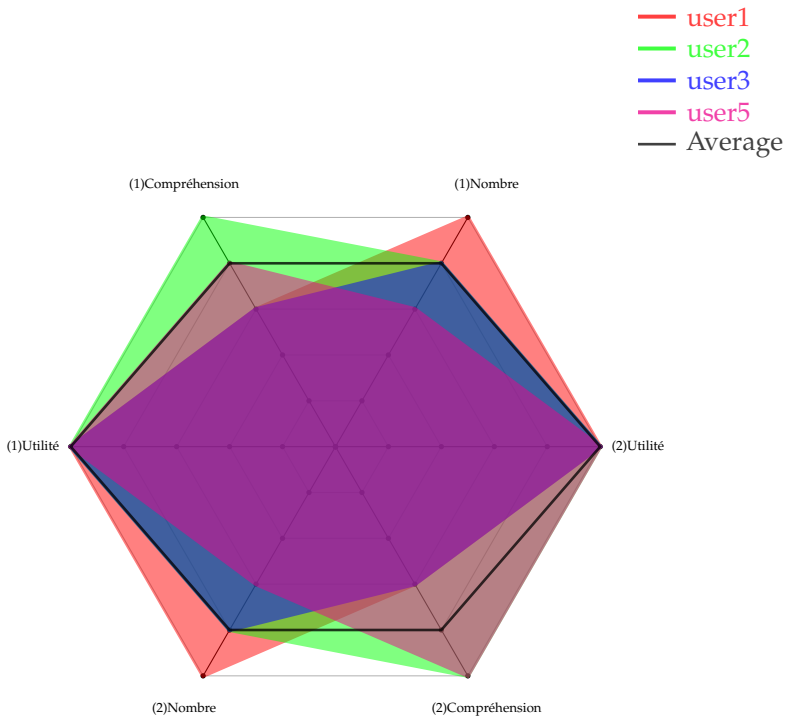


FIGURE 5.14 – Radar reprenant les critères d'évaluation des choix proposés pour la résolution des deux conflits [(1) Conflits sur les sorties et (2) Conflits sur les boutons] avec une échelle de 5

développeur.
Compréhension du conflit. La compréhension du conflit n'est pas évidente car seules les informations qui concernent le port du composant composite fonctionnelle sont fournies. Pour améliorer cela, il serait intéressant d'avoir l'ensemble des interacteurs qui sont en conflits affichés afin de faciliter la sélection de ceux-ci. De plus coupler avec cela, il serait intéressant de conserver en permanence l'affichage de la composition fonctionnelle (sous forme d'assemblage de composants) et de mettre en évidence dessus l'élément concerné par le conflit. Une autre remarque a été émise sur le fait d'avoir une pré-visualisation selon les choix effectués afin d'avoir un rendu correspondant aux choix effectués.
Résolution du conflit. Il s'avère que les possibilités de création d'éléments par réutilisation d'informations ou bien en fournissant l'ensemble des informations n'est pas évidente. Pour les utilisateurs, les possibilités les possibilités : *Choose* et *Create New One* sont redondantes. La fonction *Choose* est trop contrainte d'après l'utilisateur 5, alors que la fonction *Create New One* est trop libre. L'utilisateur 3 explicite également que ces deux fonctions devraient être regroupées en une seule et ajouter la possibilité de pouvoir modifier le libellé en plus de la sélection parmi les noms possibles. De fait, les autres éléments sont quant à eux contraints par les possibilités existantes afin d'éviter les erreurs.

Pour conclure, ces tests utilisateurs ont montré l'intérêt de l'approche et l'adéquation avec les besoins des utilisateurs puisque tous ont réussi à réaliser la même IHM avec les maquettes et l'atelier. Mais les problèmes de clarté de l'atelier n'aident pas l'utilisateur à résoudre facilement les conflits. Il est donc important de se concentrer sur ce point là afin d'améliorer la compréhension et d'ainsi permettre de faciliter l'utilisation de l'atelier.

5.3 Conclusion

Ce chapitre a présenté la validation qui a été effectuée sur l'approche et plus particulièrement sur le méta-modèle `AliasComponent` et sur l'atelier. La validation du méta-modèle `AliasComponent` s'est faite par la mise en place de transformations permettant l'abstraction des applications existantes et de la composition fonctionnelle au sein de celui-ci mais également la concrétisation du résultat obtenu par le moteur de composition. Dans le cas de l'abstraction d'un *Web Service*, l'abstraction s'effectue en utilisant des règles `ATL`. Les modèles qui résultent de ces transformations sont ainsi conformes au méta-modèle `AliasComponent` et assurent que les contraintes sont bien respectées. L'abstraction de l'IHM utilise, quant à elle, une table d'équivalences qui fait correspondre à un élément d'IHM concrète une représentation au sein du méta-modèle `AliasComponent`. Les modèles obtenus après abstraction des IHM méritent d'être retravaillés afin d'avoir une abstraction conforme à l'IHM et aux attentes. Pour ce faire, il est nécessaire d'enrichir les mécanismes d'abstraction car une simple table d'équivalences ne suffit pas à obtenir une abstraction correcte.

La transformation de concrétisation permet d'obtenir une IHM concrète afin de proposer un rendu au développeur qui réalise la composition. Cette transformation s'appuie également sur une table d'équivalences qui fait correspondre une représentation abstraite avec une représentation concrète. Une autre méthode a également été utilisée pour permettre de faire cette transformation. Elle s'appuie sur l'utilisation de transformations conditionnelles [KK04] (*Conditional Transformation*). Ces transformations permettent d'obtenir une page HTML et sont décrites en `Prolog`. Ceci a donné à la publication d'un article qui décrit la mise en place de ces transformations [BFKJ08].

Ce méta-modèle pourrait être enrichi ou complété par d'autres méta-modèles afin de conserver des informations supplémentaires sur les éléments d'origine ainsi que sur le placement de ceux-ci. L'ensemble des ces informations permettrait d'améliorer la concrétisation qui est faite. La possibilité d'avoir plusieurs méta-modèles pour conserver ces informations a été présentée dans l'article [PDJR+08].

La seconde partie de la validation s'est faite sur l'atelier développé pour réaliser la composition d'IHM. Des tests utilisateurs ont été menés afin d'évaluer l'approche proposée. Cette évaluation se focalise plus particulièrement sur la détection de conflits et la résolution de ces conflits. Les résultats des tests utilisateurs effectués sur cinq développeurs ont montré que l'approche et les solutions de résolution étaient intéressantes mais que l'outil était difficile à appréhender. Après explication sur les possibilités de résolution offertes par l'atelier, les utilisateurs ont compris la différence qu'il y a entre les différents choix proposés pour résoudre le conflit. Pour finir les débriefings ont fait ressortir des idées intéressantes afin d'améliorer l'outil et faciliter son utilisation comme : l'ajout d'une pré-visualisation selon le choix effectué, la mise en surbrillance de l'élément de la composition fonctionnelle concerné par le conflit ou encore la fusion de deux politiques de résolution (choix et création) pour n'en faire qu'une seule.

6
Conclusion et perspectives

6.1 Conclusion

Ces travaux de thèse ont montré qu'il était techniquement possible de réaliser la composition d'applications dirigée par la composition fonctionnelle, de déduire la composition d'IHM qui doit être faite et de reconstruire une application. Pour réaliser cette composition, un atelier d'aide à la composition a été développé qui s'appuie sur la formalisation et la méta-modélisation des applications à composer ainsi que de la composition fonctionnelle. Ainsi, l'intérêt de cette composition est :

- de pouvoir réutiliser l'intégralité des applications existantes grâce à l'utilisation de transformations ainsi que d'un niveau d'abstraction élevé,
- de tirer partie de l'expressivité de la composition fonctionnelle au travers de la description des flots de données et de contrôle,
- de pouvoir identifier les éléments d'IHM à conserver en fonction des besoins de la composition fonctionnelle,
- de pouvoir fournir un ensemble de choix possibles au développeur pour l'aider à résoudre un conflit,
- de recréer l'ensemble des liens d'interaction entre les IHM créées et la composition fonctionnelle afin d'obtenir une nouvelle application.

La composition réalisée au niveau des IHM ne correspond pas à une simple juxtaposition de deux ou n IHM comme dans les approches décrites en section 3.3.2.

La solution proposée répond à deux points qui ressortent de l'état qui sont : (i) l'indépendance à une technologie et (ii) la possibilité de tirer partie de l'expressivité de la composition fonctionnelle.

L'indépendance technologique est possible par l'utilisation d'un niveau d'abstraction qui permet de représenter l'ensemble des IHM et noyaux fonctionnels tout en ne conservant de ceux-ci que ce qui est nécessaire à la réalisation de la composition. Pour l'IHM, ces éléments correspondent à ceux utilisés par le noyau fonctionnel, c'est-à-dire, les éléments qui permettent de saisir des informations, de rendre les informations à l'utilisateur ou de déclencher l'exécution d'une opération. Pour le noyau fonctionnel, l'ensemble des prototypes des opérations est conservé et pour la composition, le flot de contrôle et de données ainsi que le prototype des opérations. L'ensemble de ces informations est décrit au sein du méta-modèle `AliasComponent`. Ce méta-modèle s'inspire des

approches à composants en représentant le noyau fonctionnel (resp. les IHM) comme des composants fonctionnels (resp. d'IHM). L'assemblage de ces composants, pour former une application, est fait au travers de deux types de liens que sont les liens d'événements (décrivant le déclenchement d'opération) et les liens de données (décrivant les échanges de données entre l'IHM et le noyau fonctionnel). La composition fonctionnelle est quant à elle décrite par un composant composite fonctionnel qui regroupe l'ensemble des noyaux fonctionnels (sous-composants fonctionnels) des applications composées. De la même manière que pour décrire une application, des liens de données et d'événement entre le composant composite fonctionnel et ses sous-composants mais également entre sous-composants permettent de décrire le flot de données et de contrôle. `AliasComponent` se distingue des autres approches à composants en distinguant : (i) des ports de données et des ports d'événements et (ii) des liens de données et d'événement. Ceci est nécessaire pour réaliser la composition d'application.
La possibilité de tirer partie de l'expressivité de la composition fonctionnelle se fait au travers de l'analyse du flot de données et de contrôle. En effet, le flot de données décrit l'ensemble des échanges qu'il y a entre les opérations. Il est ainsi possible de décrire : (i) des partages de données entre plusieurs opérations, (ii) des transferts de données d'une opération à une autre et (iii) des fusions de données en provenance de plusieurs opérations. Cette analyse du flot de données est un point important du processus de composition dans la déduction des éléments d'IHM à conserver.

Les réponses et apports de cette thèse dans les domaines des IHM et du logiciel sont :

- la mise en place du méta-modèle `AliasComponent` qui permet d'unifier la représentation que l'on a des IHM et du noyau fonctionnel tout en étant indépendant d'une technologie donnée. Ce méta-modèle tire également parti des travaux effectués sur les modèles de tâches en IHM et ceux de la composition fonctionnelle, afin de faire ressortir à la fois le flot de données et le flot de contrôle, distinction nécessaire pour l'identification des éléments à conserver lors de la composition.
- le processus de composition qui permet la réutilisation et la composition d'applications existantes. Il est ainsi possible de réutiliser au maximum les informations disponibles tant au niveau des IHM que des interactions entre les IHM et le noyau fonctionnel mais également au sein de la composition fonctionnelle. De l'ensemble de ces informations, il construit une nouvelle application dont les IHM créées sont composées d'éléments présents dans les IHM existantes et d'éléments sélectionnés par le développeur lorsque le moteur de composition n'a pu décider par lui-même.

Ces travaux de thèse ont montré l'intérêt d'avoir une représentation abstraite des IHM (comme décrite au sein du CRF) et des interactions qu'il peut y avoir avec le noyau fonctionnel. De la même manière que l'on peut décrire des *Web Services* au travers de l'utilisation de WSDL, il serait intéressant d'utiliser une description similaire pour les IHM. Ainsi, on aurait :

- un détachement technologique : l'interface cache la technologie qui a été utilisée pour développer l'IHM et il est ainsi possible de la manipuler sans avoir à connaître son fonctionnement ;
- un mécanisme d'assemblage IHM–NF : de la même manière que l'on a les orchestrations de *Web Services*, celles-ci, pourraient être utilisées afin de pouvoir décrire le flot de contrôle et de données entre les IHM et les *Web Services* ;
- un mécanisme de composition d'IHM : basé sur le même principe que les orchestrations, mais cette fois-ci dédiée aux IHM. Ces mécanismes permettraient de composer les IHM entre elles et de décrire de la même manière que dans la partie fonctionnelle : des partages/fusions de données, parallélisme, séquence, condition... Ainsi, on aurait la possibilité de décrire le *workflow* des IHM.

6.2 Perspectives

Ce qui suit présente les perspectives possibles de ces travaux. Dans un premier temps, les perspectives à court terme puis les perspectives à long terme.

6.2.1 Perspectives à court terme

Amélioration de l'atelier de composition

Les retours utilisateurs ont montré que l'atelier devait être amélioré afin de faciliter son utilisation et aider à la compréhension des choix offerts au développeur pour résoudre les conflits détectés lors du processus de composition. En effet, la partie de l'atelier qui permet de faire cette résolution reste relativement simple et ne met pas suffisamment en avant l'élément concerné au niveau de la composition fonctionnelle. Les attributs des éléments d'IHM qui sont différents ne ressortent pas non plus. Il serait donc intéressant dans un premier temps de prendre en considération les remarques qui ont été faites, c'est-à-dire :

- ajout de la visualisation du conflit sur l'assemblage correspondant à la composition fonctionnelle afin de connaître les éléments mis en jeu dans ce conflit. Ceci afin d'aider aux choix que doit faire le développeur.
- ajout de la visualisation des éléments en conflits afin de visualiser pourquoi il y a un conflit et quels sont les critères qui font que l'on a un conflit.
- ajout d'une prévisualisation afin que le développeur puisse se rendre compte des choix qu'il est en train de faire et qu'il ne soit pas obligé d'attendre la fin du processus de composition pour avoir un rendu visuel de l'IHM résultante.
- fusion des possibilités : *Choose* et *Create* pour n'en former qu'une seule. En effet, il n'est pas nécessaire que le développeur ait le choix parmi des éléments qui ne sont pas corrects. Cela passe également par l'ajout d'une possibilité d'édition sur le nom associé (pour modifier le label affiché à côté de l'élément d'IHM).

Après ces corrections au sein de l'atelier, il serait intéressant d'effectuer de nouveaux tests utilisateurs pour permettre de valider les modifications et vérifier que celles-ci étaient bien nécessaires pour une meilleure compréhension du processus de composition et une diminution des interrogations que le développeur peut avoir au moment de la résolution des conflits.

Conservation des choix effectués par le développeur

Lors de la résolution des conflits, le développeur doit effectuer un certain nombre de choix. Dans le cas où il y a de nombreux conflits, ce travail est long et fastidieux. Une extension possible à l'atelier serait de conserver l'ensemble des choix effectués par le développeur en vue de rejouer la composition. Dans le cas où le développeur souhaiterait rejouer la composition, l'ensemble des choix précédemment faits pourrait être appliqué et la possibilité d'en modifier certains lui être offerte. Ainsi, le développeur pourrait se concentrer sur une sous-partie des conflits à traiter afin d'obtenir un nouveau résultat de composition.

De la même manière que l'on peut conserver les choix effectués par le développeur pour rejouer une composition, ceux-ci pourraient également être associés aux applications composées et à la composition fonctionnelle réalisée. Ainsi, si le développeur souhaite réaliser une composition identique sur des applications identiques à celles précédemment composées, les choix effectués auparavant pourraient être réappliqués directement sans avoir besoin de demander à nouveau au développeur de résoudre l'ensemble des conflits. Une grille récapitulative des choix effectués permettrait ainsi d'en modifier une sous-partie.

6.2.2 Perspectives à long terme

Amélioration de la concrétisation des IHM résultantes

La concrétisation que l'on a des IHM obtenues suite à la réalisation de la composition est simple puisque le processus d'abstraction n'a pas conservé l'intégralité des informations présentes au sein des IHM existantes. Pour permettre d'améliorer cette concrétisation, deux possibilités peuvent être envisagées : (i) l'enrichissement des modèles existants avec l'amélioration des transformations d'abstraction et/ou (ii) le rapprochement avec des modèles existants dédiés à la description des IHM.

(i) Enrichissement des modèles existants et amélioration des transformations d'abstraction. Une première possibilité afin d'améliorer la concrétisation des IHM est d'enrichir les modèles existants avec d'autres modèles tels que ceux présentés dans [PDJR+08]. Cela permettrait de conserver plus d'informations sur les IHM abstraites telles que les caractéristiques des éléments concrets ainsi que le placement des éléments. Ainsi, ces informations pourraient être utilisées par la suite au moment de la concrétisation afin d'améliorer le rendu de l'IHM finale.

Une autre possibilité pour enrichir les modèles existants consisterait à prendre en considération d'autre support d'interaction et viserait donc à enrichir le modèle d'interaction afin de pouvoir gérer des supports tactiles ou vocaux.

Le fait d'enrichir les modèles nécessite de modifier les transformations d'abstraction. Dans un premier temps, il serait intéressant d'améliorer ces transformations afin d'obtenir un résultat correct puis de les enrichir pour permettre de compléter l'ensemble des modèles ajoutés pour améliorer la concrétisation.

Les mécanismes d'abstraction des IHM présentés en section 5.1.1 basés sur l'utilisation d'une table d'abstraction ne permettent pas d'obtenir un résultat correct. Il est donc nécessaire d'améliorer cette transformation. Pour ce faire, plusieurs possibilités sont envisageables : (i) l'étude des interactions entre l'IHM et le noyau fonctionnel et (ii) l'étude du placement des éléments. L'étude des interactions entre l'IHM et le noyau fonctionnel permettrait d'assurer que les éléments d'IHM abstraits sont bien ceux que l'on souhaite puisqu'ils sont liés au noyau fonctionnel mais également de s'assurer que les éléments sont abstraits dans la bonne catégorie (entrée, sortie ou événement). L'étude du placement des éléments permettrait de déterminer d'une part le placement des éléments les uns par rapport aux autres pour compléter un modèle de placement des éléments mais également de pouvoir déterminer qu'un label désigne un *widget* que l'on doit abstraire. Ainsi, on pourrait renseigner l'attribut *name* pour chacun des éléments abstraits. Utiliser ces deux possibilités avec la table d'abstraction permettraient d'obtenir un meilleur résultat nécessitant moins de travail de la part du développeur.

L'enrichissement des transformations d'abstraction est nécessaire pour permettre d'obtenir les nouvelles informations mises en place par l'enrichissement des modèles. Il serait également intéressant d'enrichir ou de modifier les transformations existantes afin de prendre en considération d'autres types d'IHM comme les IHM post-WIMP.

(ii) Rapprochement avec des modèles existants. L'intérêt de se rapprocher de travaux tels que ceux d'USIXML ou bien encore `MARIA` [PSS09] est de pouvoir viser la plasticité des IHM et également de pouvoir réutiliser l'ensemble de leurs outils de transformations. Ce rapprochement pourrait se faire au travers de l'usage de la collaboration de modèles [ODPRK08] en conservant le méta-modèle `AliasComponent` comme méta-modèle central et en le faisant collaborer avec les modèles d'USIXML ou de `MARIA`. Il permettrait ainsi :

– d'obtenir des IHM finales plastiques adaptables à différents supports d'exécution sans avoir à d'écrire une nouvelle transformation spécifique ;

– d'utiliser les outils déjà développés pour réaliser des transformations dans différents langages cibles et d'ainsi éviter de décrire des transformations *ad-hoc* pour chacune des technologies que l'on souhaite atteindre.

Amélioration des mécanismes de compositions

Pour le moment, les mécanismes de composition s'appuient exclusivement sur la description des applications à composer ainsi que sur la description de la composition fonctionnelle. Afin d'améliorer les mécanismes de composition mis en œuvre, il serait intéressant de réaliser la composition en partant des IHM afin de pouvoir augmenter le nombre d'éléments que l'on peut fusionner ou supprimer. Un autre point intéressant serait la conservation des modèles à l'exécution pour réaliser la composition à l'exécution et non plus durant la phase de développement.

(i) Réalisation de la composition dirigée par les IHM. L'approche qui a été présentée réalise la composition d'IHM dirigée par la composition fonctionnelle. Le but serait donc de réutiliser les descriptions d'applications ainsi que des propriétés mises en place pour permettre de compléter la composition d'IHM réalisée à partir de la composition fonctionnelle. Cela permettrait de proposer un ensemble d'éléments qui peuvent être fusionnés. Pour ce faire, il serait possible de se rapprocher des travaux sur la composition d'IHM enrichis par la connaissance que l'on a sur le noyau fonctionnel sous-jacent. Le développeur aurait ainsi l'ensemble des informations à sa disposition pour réaliser ces choix de composition.

Une autre possibilité serait d'utiliser les travaux basés sur les arbres de tâches. En effet, l'arbre de tâches est proche de ce que l'on peut retrouver au sein des orchestrations de *Web Services* pour le fonctionnel. Cela permettrait ainsi d'ajouter la notion de temporalité et d'avoir la possibilité de décrire les enchaînements d'écran que l'on ne peut pas décrire dans les compositions fonctionnelles.

(ii) Conservation des modèles à l'exécution. L'intérêt de pouvoir conserver les modèles à l'exécution est de pouvoir : (i) améliorer les transformations que l'on peut avoir des IHM afin de pouvoir obtenir une IHM finale sans intervention d'un designer et (ii) réaliser la composition à l'exécution et donc ne plus être forcément destiné aux développeurs. La réalisation de la composition à l'exécution permettrait de pouvoir cibler d'autres domaines d'applications tels que : l'informatique ambiante ou la domotique.

Par contre, il serait intéressant de revoir les mécanismes de compositions afin de les rendre le plus accessible possible pour ne plus être dédiés aux développeurs mais qu'un utilisateur final puisse lui même réaliser la composition ou bien encore qu'il puisse (dans le cas de l'informatique ambiante) spécifier les choix qui permettent de résoudre les conflits détectés et ainsi obtenir l'IHM dont il a besoin. Pour faciliter les choix offerts à l'utilisateur, il serait possible d'ajouter de la sémantique sur les éléments qu'il doit manipuler afin de leur donner du sens. Une autre possibilité serait également de revoir l'IHM de l'outil permettant la prise de décision afin de la simplifier et de contraindre les choix offerts pour éviter de perdre l'utilisateur.

Bibliographie

[AD67] John Annett and Keith D. Duncan. Task analysis and training design. *Occupational Psychology*, 41 :211–221, 1967.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11) :823–843, November 1983.

[Ann03] John Annett. *Hierarchical Task Analysis*, chapter 2, pages 17–35. Handbook of cognitive task design. CRC Press, June 2003.

[Arc07] Open Service Oriented Architecture. Service Component Architecture — Assembly Model Specification. Version 1.0. http://osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf, March 2007.

[BB08] Stefan Betermieux and Birgit Bomsdorf. Task-Driven Composition of Web User Interfaces. In *Proceedings of 7th International Conference on Computer-Aided Design of User Interfaces*, CADUI'08, pages 233–244, Albacete, Spain, June 2008. Springer London.

[BC92] Leonard J. Bass and Joëlle Coutaz. A metamodel for the runtime architecture of an interactive system : the UIMS tool developers workshop. *SIGCHI Bulletin*, 24(1) :32–37, January 1992.

[BCS02] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming*, WCOP'02, Málaga, Spain, June 2002. Springer Berlin / Heidelberg.

[BFKJ08] Pascal Bihler, Merlin Fotsing, Günter Kniesel, and Cédric Joffroy. Using Conditional Transformations for Semantic User Interface Adaptation. In *Proceedings of the 10th International Conference on Information Integration and Web-based Application and Services*, iiWAS'08, pages 677–680, Linz, Austria, November 2008. ACM.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. http://www.w3.org/TR/ws-arch/, February 2004. W3C.

[BLT07] Grégory Bourguin, Arnaud Lewandowski, and Jean-Claude Tarby. Defining task oriented components. In *Proceedings of the 6th international conference on Task models and diagrams for user interface design*, TAMODIA'07, pages 170–183, Toulouse, France, November 2007. Springer-Verlag.

[Bom07] Birgit Bomsdorf. The WebTaskModel approach to web process modelling. In *Proceedings of the 6th international conference on Task models and diagrams for user interface design*, TAMODIA'07, pages 240–253, Toulouse, France, November 2007. Springer-Verlag.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3) :289–308, June 2003.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, CA, USA, October 2003.

[CJL09] Benjamin Caramel, Cédric Joffroy, and Michael Laguerre. De la composition de services à la composition d'Interfaces Homme-Machine. In *Actes de la 21ème Conférence Francophone sur l'Interaction Homme-Machine*, IHM'09, pages 65–74, Grenoble, France, October 2009. ACM.

[CL95] Donald D. Cowan and Carlos J. P. Lucena. Abstract Data Views : An Interface Specification Concept to Enhance Design for Reuse. *IEEE Transactions on Software Engineering*, 21(3) :229–243, March 1995.

[Cou87] Joëlle Coutaz. PAC : an Object Oriented Model for Implementing User Interfaces. *SIGCHI Bulletin*, 19(2) :37–41, October 1987.

[DPJO+09] Anne-Marie Dery-Pinna, Cédric Joffroy, Audrey Occello, Philippe Renevier, and Michel Riveill. L'Ingénierie Dirigée par les Modèles au cœur d'un Framework d'aide à la composition d'interfaces utilisateurs. In *Actes de la 5 ème journée sur l'Ingénierie Dirigée par les Modèles*, IDM'09, pages 131–146, Nancy, France, March 2009.

[Eng97] Robert Englander. *Developing Java Beans*. O'Reilly Media, 1997.

[Erl05] Thomas Erl. *Service-Oriented Architecture : Concepts, Technology, and Design*. Prentice Hall, 2005.

[FHSS09] Marius Feldmann, Gerald Hübsch, Thomas Springer, and Alexander Schill. Improving Task-driven Software Development Approaches for Creating Service-Based Interactive Applications by Using Annotated Web Services. In *Proceedings of the 2009 Fifth International Conference on Next Generation Web Services Practices*, NWESP'09, pages 94–97, Washington, DC, USA, September 2009. IEEE Computer Society.

[FJN+09] Marius Feldmann, Jordan Janeiro, Tobias Nestler, Gerald Hübsch, Uwe Jugel, André Preussner, and Alexander Schill. An Integrated Approach for Creating Service-Based Interactive Applications. In *Proceedings of the 12th IFIP TC13 International Conference on Human-Computer Interaction*, INTERACT'09, pages 896–899, Uppsala, Sweden, August 2009. Springer-Verlag.

[GCF08a] Yoann Gabillon, Gaëlle Calvary, and Humbert Fiorino. Composing interactive systems by planning. In *Proceedings of the 4th French-speaking conference on Mobility and ubiquity computing*, UbiMob'08, pages 37–40, Saint-Malo, France, May 2008. ACM.

[GCF08b] Yoann Gabillon, Gaëlle Calvary, and Humbert Fiorino. L'IDM passerelle entre IHM et planification pour la composition dynamique de systèmes interactifs. In *Proceedings of 4ème Journées sur l'Ingénierie Dirigée par les Modèles*, IDM'08, pages 51–56, Mulhouse, France, June 2008.

[GHM+07] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1 : Messaging Framework (Second Edition). http://www.w3.org/TR/soap12-part1/, April 2007. W3C.

[GRUD07] Jeronimo Ginzburg, Gustavo Rossi, Matias Urbieta, and Damiano Distante. Transparent interface composition in web applications. In *Proceedings of the 7th international conference on Web engineering*, ICWE'07, pages 152–166, Como, Italy, July 2007. Springer-Verlag.

[HC01] George T. Heineman and William T. Councill. *Component-based software engineering : putting the pieces together*. Addison-Wesley Professional, 2001.

[HSH90] H. Rex Hartson, Antonio C. Siochi, and D. Hix. The UAN : a user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, 8(3) :181–203, July 1990.

[HTL+08] Vincent Hourdin, Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. SLCA, composite services for ubiquitous computing. In *Proceedings of the International Conference on Mobile Technology, Applications, and Systems*, Mobility'08, pages 11 :1–11 :8, Yilan, Taiwan, September 2008. ACM.

[IBM06] IBM. Service Component Architecture. http://www-128.ibm.com/developerworks/library/specification/ws-sca/, 2006.

[IJH+09] Paoli Izquierdo, Jordan Janeiro, Gerald Hübsch, Thomas Springer, and Alexander Schill. An annotation tool for enhancing the user interface generation process for services. In *Proceedings of the 19th International Crimean Conference, Microwave & Telecommunication Technology*, CriMiCo'09, pages 372–374, Sevastopol, Crimea, Ukraine, September 2009. IEEE.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL : a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA'06, pages 719–720, Portland, Oregon, USA, October 2006. ACM.

[JCDPR11] Cédric Joffroy, Benjamin Caramel, Anne-Marie Dery-Pinna, and Michel Riveill. When the Functional Composition Drives the User Interfaces Composition : Process and Formalization. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS'11, Pisa, Italy, June 2011. ACM. A paraître.

[JK94] Bonnie E. John and David E. Kieras. The GOMS Family of Analysis Techniques : Tools for Design and Evaluation. Technical report, Human–Computer Interaction Institute, Pittsburgh, PA, USA, 1994.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the 8th MoDELS International Workshop on Model Transformations in Practice*, MTiP'05, pages 128–138, Montego Bay, Jamaica, October 2005.

[KK04] Günter Kniesel and Helge Koch. Program-Independent Composition of Conditional Program Transformations. Technical report, Institute for Computer Science III at the University of Bonn, Germany, February 2004.

[KMW03] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-Oriented Composition in BPEL4WS. In *Proceedings of the 20th Internation World Wide Web Conference (Alternate Papers Track)*, WWW'03, Budapest, Hungary, May 2003.

[LHR+07] Sophie Lepreux, Anas Hariri, José Rouillard, Dimitri Tabary, Jean-Claude Tarby, and Christophe Kolski. Towards multimodal user interfaces composition based on UsiXML and MBD principles. In *Proceedings of the 12th International Conference on Human-Computer Interaction*, HCI'07, pages 134–143, Beijing, China, July 2007. Springer-Verlag.

[LLB07] Arnaud Lewandowski, Sophie Lepreux, and Grégory Bourguin. Tasks models merging for high-level component composition. In *Proceedings of the 12th International Conference on Human-Computer Interaction*, HCI'07, pages 1129–1138, Beijing, China, July 2007. Springer-Verlag.

[LV06] Sophie Lepreux and Jean Vanderdonckt. Towards A Support of User Interface Design By Composition Rules. In *Proceedings of 6th International Conference on Computer-Aided Design of User Interfaces*, CADUI'06, pages 231–244, Bucharest, Romania, June 2006. Springer London.

[LVM+04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. USIXML : A User Interface Description Language for Context-Sensitive User Interfaces. In *Proceeding of the ACM AVI'2004 Developing User Interfaces with XML : Advances on User Interface Description Languages Workshop*, pages 55–62, Gallipoli, Italy, May 2004. ACM.

[LVM06] Sophie Lepreux, Jean Vanderdonckt, and Benjamin Michotte. Visual Design of User Interfaces by (De)composition. In *Proceedings of the 13th International Conference on Interactive systems : Design, specification, and verification*, DSVIS'06, pages 157–170, Dublin, Ireland, July 2006. Springer-Verlag.

[MBH+06] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S : Semantic Markup for Web Services. http://www.ai.sri.com/daml/services/owl-s/1.2/overview/, March 2006.

[Mer06] Duane Merrill. Mashups : The new breed of Web app—An introduction to mashups. Technical report, IBM, August 2006. http://www.ibm.com/developerworks/xml/library/x-mashups.html.

[MM04] Nikola Milanovic and Miroslaw Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6) :51–59, November 2004.

[MPS02] Giulio Mori, Fabio Paternò, and Carmen Santoro. CTTE : Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering*, 28(8) :797–813, August 2002.

[MR09] Jim Marino and Michael Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 2009.

[NFPS09] Tobias Nestler, Marius Feldmann, Andre Preuÿner, and Alexander Schill. Service Composition at the Presentation Layer using Web Service Annotations. In *First International Workshop on Lightweight Integration on the Web*, ComposableWeb'09, pages 63–68, San Sebastian, Spain, June 2009.

[Obj08] Objectweb Consortium. The Fractal Component Model. http://fractal.objectweb.org/, 2008.

[ODPRK08] Audrey Occello, Anne-Marie Dery-Pinna, Michel Riveill, and Günter Kniesel. Managing Model Evolution Using the CCBM Approach. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS'08, pages 453–462, Dublin, Ireland, April 2008. IEEE.

[OJDP10a] Audrey Occello, Cédric Joffroy, and Anne-Marie Dery-Pinna. Experiments in Model Driven Composition of User Interfaces. In *Proceedings of the 10th IFIP International Conference on Distributed Applications and Interoperable Systems*, DAIS'10, pages 98–111, Amsterdam, Netherlands, June 2010. Springer-Verlag.

[OJDP+10b] Audrey Occello, Cédric Joffroy, Anne-Marie Dery-Pinna, Philippe Renevier, and Michel Riveill. Metamodeling user interfaces and services for composition considerations. In *Proceedings of the 19th International Conference on Software Engineering and Data Engineering*, SEDE'10, pages 33–38, San Francisco, California, USA, June 2010. ISCA.

[OMG07] OMG. Data Distribution Service for Real-time Systems, January 2007.

[OMG08] OMG. The Common Object Request Broker Architecture : Core Specification. Version 3.1, January 2008.

[OMG09] OMG. UML 2 Specification, February 2009.

[Pat00] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 1st edition, January 2000.

[PDF03] Anne-Marie Pinna-Dery and Jérémy Fierstone. Component model and programming : a first step to manage Human Computer Interaction Adaptation. In *Proceedings of the 5th International Symposium on Human Computer Interaction with Mobile*

Devices and Services, Mobile HCI'03, pages 456–460, Udine, Italy, September 2003. Springer-Verlag.

[PDJR+08] Anne-Marie Pinna-Dery, Cédric Joffroy, Philippe Renevier, Michel Riveill, and Christophe Vergoni. ALIAS : A Set of Abstract Languages for User Interface Assembly. In *Proceedings of the 12th IASTED International Conference on Software Engineering and Applications*, SEA'08, pages 77–82, Orlando, Florida, USA, November 2008. ACTA Press.

[Pie09] Stefan Pietschmann. Model-Driven Development and Runtime Platform for Adaptive Composite Web Applications. *International Journal on Advances in Internet Technology*, 2(4) :277–288, 2009.

[PMM97] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees : A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the 6th IFIP TC13 International Conference on Human-Computer Interaction*, INTERACT'97, pages 362–369, Sydney, Australia, July 1997. Chapman & Hall.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA : A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction*, 16(4) :19 :1–19 :30, November 2009.

[PVM09] Stefan Pietschmann, Martin Voigt, and Klaus Meissner. Dynamic Composition of Service-Oriented Web User Interfaces. In *Proceedings of the 4th International Conference on Internet and Web Applications and Services*, ICIW'09, pages 217–222, Venice/Mestre, Italy, May 2009. IEEE Computer Society.

[Ree79] Trygve M. H. Reenskaug. MVC XEROX PARC 1978–1979. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, 1979.

[SGC+07] Jean-Sebastien Sottet, Vincent Ganneau, Gaëlle Calvary, Joëlle Coutaz, Jean-Marie Favre, and Rachel Demumieux. Model-Driven Adaptation for Plastic User Interfaces. In *Proceedings of the 11th IFIP TC13 International Conference on Human-Computer Interaction*, INTERACT'07, pages 397–410, Rio de Janeiro, Brasil, September 2007. Springer-Verlag.

[Sot08] Jean-Sébastien Sottet. *Méga–IHM : malléabilité des Interfaces Homme–Machine dirigées par les modèles*. PhD thesis, Université Joseph Fourier, Laboratoire d'Informatique de Grenoble, Grenoble, France, 2008.

[SR98] Daniel Schwabe and Gustavo Rossi. An object oriented approach to Web-based applications design. *Theory and Practice of Object Systems - Special issue objects, databases, and the WWW*, 4(4) :207–225, October 1998.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2nd edition, November 2002.

[TC99] David Thevenin and Joëlle Coutaz. Adaptation and Plasticity of User Interfaces. In *Workshop on Adaptive Design of Interactive Multimedia Presentations for Mobile Users*, i3 Spring Days, Sitges, Spain, March 1999.

[THEC08] Wei-Tek Tsai, Qian Huang, Jay Elston, and Yinong Chen. Service-Oriented User Interface Modeling and Composition. In *Proceedings of the IEEE International Conference on e-Business Engineering*, ICEBE'08, pages 21–28, Xi'an, China, October 2008. IEEE Computer Society.

[TLR+09] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, and Michel Riveill. Lightweight Service Oriented Architecture for Pervasive Computing. *International Journal of Computer Science Issues*, 4(1) :1–9, September 2009.

[WS-07] *Web Services Business Process Execution Language Version 2.0*. OASIS Standard, April 2007.

Annexes

A

Description des applications et des compositions

A.1 Schémas des applications

A.1.1 *SocialInsurance Application*

Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
            oasis-200401-wss-wssecurity-utility-1.0.xsd"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace="http://rainbow.i3s.unice.fr/socialinusrance"
 name="SocialInsuranceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://rainbow.i3s.unice.fr/socialinusrance"
         schemaLocation="http://localhost:14318/SocialTest/SocialInsuranceService?xsd=1">
      </xsd:import>
    </xsd:schema>
  </types>
  <message name="getByName">
    <part name="parameters" element="tns:getByName"></part>
  </message>
  <message name="getByNameResponse">
    <part name="parameters" element="tns:getByNameResponse"></part>
  </message>
  <message name="getByCard">
    <part name="parameters" element="tns:getByCard"></part>
  </message>
  <message name="getByCardResponse">
    <part name="parameters" element="tns:getByCardResponse"></part>
  </message>
  <portType name="socialinsurance">
    <operation name="getByName">
      <input message="tns:getByName"></input>
      <output message="tns:getByNameResponse"></output>
    </operation>
    <operation name="getByCard">
```

```
      <input message="tns:getByCard"></input>
      <output message="tns:getByCardResponse"></output>
    </operation>
  </portType>
  <binding name="socialInsurancePortBinding" type="tns:socialinsurance">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document">
    </soap:binding>
    <operation name="getByName">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
    <operation name="getByCard">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
  </binding>
  <service name="SocialInsuranceService">
    <port name="socialInsurancePort" binding="tns:socialInsurancePortBinding">
      <soap:address location="http://localhost:14318/SocialTest/SocialInsuranceService">
      </soap:address>
    </port>
  </service>
</definitions>
```

Listing A.1 – WSDL pour le service de `SocialInsuranceService`

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="1.0" targetNamespace="http://rainbow.i3s.unice.fr/socialinusrance">

  <xs:element name="getByCard" type="tns:getByCard" />
  <xs:element name="getByCardResponse" type="tns:getByCardResponse" />
  <xs:element name="getByName" type="tns:getByName" />
  <xs:element name="getByNameResponse" type="tns:getByNameResponse" />

  <xs:complexType name="getByCard">
    <xs:sequence>
      <xs:element name="cardID" type="xs:long" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="getByCardResponse">
    <xs:sequence>
      <xs:element name="return" type="tns:socialInsuranceInformation" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="socialInsuranceInformation">
    <xs:sequence>
      <xs:element name="address" type="xs:string" minOccurs="0" />
      <xs:element name="date" type="xs:dateTime" minOccurs="0" />
      <xs:element name="email" type="xs:string" minOccurs="0" />
      <xs:element name="firstName" type="xs:string" minOccurs="0" />
      <xs:element name="insuranceNumber" type="xs:long" />
      <xs:element name="lastName" type="xs:string" minOccurs="0" />
      <xs:element name="referee" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="getByName">
    <xs:sequence>
      <xs:element name="firstName" type="xs:string" minOccurs="0" />
      <xs:element name="lastName" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="getByNameResponse">
    <xs:sequence>
      <xs:element name="return" type="tns:socialInsuranceInformation" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.2 – Schéma de données pour le Web Service `SocialInsuranceService`

IHM associée

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  minWidth="955" minHeight="600"
  xmlns:socialinsuranceservice="services.socialinsuranceservice.*"
  width="317" height="336">
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      import mx.rpc.events.ResultEvent;

      protected function button_clickHandler(event:MouseEvent):void
      {
        getByCardResult.token = socialInsuranceService.getByCard(parseInt(carIdInput.text));
      }

      protected function Social_result(evt:ResultEvent):void
      {
        FirstNameOutput.text = getByCardResult.lastResult.firstName;
        LastNameOutput.text = getByCardResult.lastResult.lastName;
        EmailOutput.text = getByCardResult.lastResult.email;
        InsuranceNumberOutput.text = getByCardResult.lastResult.insuranceNumber;
        RefereeOutput.text = getByCardResult.lastResult.referee;
        AddressOutput.text = getByCardResult.lastResult.address;
        BirthdayOutput.text = getByCardResult.lastResult._date;
      }
    ]]>
  </fx:Script>
  <fx:Declarations>
    <s:CallResponder id="getByCardResult"/>
    <socialinsuranceservice:SocialInsuranceService id="socialInsuranceService"
       fault="Alert.show(event.fault.faultString + '\n' + event.fault.faultDetail)"
       showBusyCursor="true" result="Social_result(event)"/>

  </fx:Declarations>
  <s:Label x="10" y="33" text="CardID"/>
  <s:TextInput x="69" y="28" id="carIdInput"/>
  <s:Button x="213" y="28.5" label="getByCardID" id="button"
     click="button_clickHandler(event)"/>
  <mx:HRule x="0" y="58" width="318"/>
  <s:Label x="10" y="79" text="FirstName: "/>
  <s:Label x="10" y="113" text="LastName: "/>
  <s:Label x="9" y="147" text="Email: "/>
  <s:Label x="9" y="184" text="Birthday: "/>
  <s:Label x="10" y="222" text="InsuranceNumber: "/>
  <s:Label x="10" y="261" text="Referee: "/>
  <s:Label x="10" y="299" text="Address: "/>
  <s:Label x="82" y="79" text="-" id="FirstNameOutput"/>
  <s:Label x="80" y="113" text="-" id="LastNameOutput"/>
  <s:Label x="79" y="147" text="-" id="EmailOutput"/>
  <s:Label x="79" y="184" text="-" id="BirthdayOutput"/>
  <s:Label x="122" y="222" text="-" id="InsuranceNumberOutput"/>
  <s:Label x="80" y="261" text="-" id="RefereeOutput"/>
  <s:Label x="80" y="299" text="-" id="AddressOutput"/>
</s:Application>
```

Listing A.3 – IHM associée à l'opération `getByCard`

A.1.2 *Business Application*

Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
            oasis-200401-wss-wssecurity-utility-1.0.xsd"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://rainbow.i3s.unice.fr/business"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace="http://rainbow.i3s.unice.fr/business" name="BusinessService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://rainbow.i3s.unice.fr/business"
          schemaLocation="http://localhost:14318/BusinessTest/BusinessService?xsd=1">
      </xsd:import>
    </xsd:schema>
  </types>
  <message name="getAddresses">
    <part name="parameters" element="tns:getAddresses"></part>
  </message>
  <message name="getAddressesResponse">
    <part name="parameters" element="tns:getAddressesResponse"></part>
  </message>
  <message name="getBusinessInfo">
    <part name="parameters" element="tns:getBusinessInfo"></part>
  </message>
  <message name="getBusinessInfoResponse">
    <part name="parameters" element="tns:getBusinessInfoResponse"></part>
  </message>
  <portType name="business">
    <operation name="getAddresses">
      <input message="tns:getAddresses"></input>
      <output message="tns:getAddressesResponse"></output>
    </operation>
    <operation name="getBusinessInfo">
      <input message="tns:getBusinessInfo"></input>
      <output message="tns:getBusinessInfoResponse"></output>
    </operation>
  </portType>
  <binding name="businessPortBinding" type="tns:business">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document">
    </soap:binding>
    <operation name="getAddresses">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
    <operation name="getBusinessInfo">
      <soap:operation soapAction=""></soap:operation>
      <input>
        <soap:body use="literal"></soap:body>
      </input>
      <output>
        <soap:body use="literal"></soap:body>
      </output>
    </operation>
  </binding>
  <service name="BusinessService">
    <port name="businessPort" binding="tns:businessPortBinding">
      <soap:address location="http://localhost:14318/BusinessTest/BusinessService">
      </soap:address>
    </port>
  </service>
</definitions>
```

Listing A.4 – WSDL pour le service de `BusinessService`

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://rainbow.i3s.unice.fr/business"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="1.0" targetNamespace="http://rainbow.i3s.unice.fr/business">

  <xs:element name="getAddresses" type="tns:getAddresses" />
  <xs:element name="getAddressesResponse" type="tns:getAddressesResponse" />
  <xs:element name="getBusinessInfo" type="tns:getBusinessInfo" />
  <xs:element name="getBusinessInfoResponse" type="tns:getBusinessInfoResponse" />

  <xs:complexType name="getAddresses">
    <xs:sequence>
      <xs:element name="fullName" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="getAddressesResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" nillable="true"
                  minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="getBusinessInfo">
    <xs:sequence>
      <xs:element name="fullName" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="getBusinessInfoResponse">
    <xs:sequence>
      <xs:element name="return" type="tns:businessInformation" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="businessInformation">
    <xs:sequence>
      <xs:element name="address" type="xs:string" nillable="true"
                  minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="building" type="xs:string" nillable="true"
                  minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="email" type="xs:string" minOccurs="0" />
      <xs:element name="fullName" type="xs:string" minOccurs="0" />
      <xs:element name="office" type="xs:string" nillable="true"
                  minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="position" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing A.5 – Schéma de données pour le Web Service `BusinessService`

IHM associée

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx"
               minWidth="955" minHeight="600"
               xmlns:businessservice="services.businessservice.*"
               width="643" height="330">
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      import mx.rpc.events.ResultEvent;
      import mx.collections.ArrayCollection;

      [Bindable] private var addressList:ArrayCollection = new ArrayCollection();
      [Bindable] private var officeList:ArrayCollection = new ArrayCollection();
      [Bindable] private var buildingList:ArrayCollection = new ArrayCollection();

      protected function button_clickHandler(event:MouseEvent):void
      {
```

```
        getBusinessInfoResult.token = businessService.getBusinessInfo(FullNameInput.text);
      }

      protected function Business_result(evt:ResultEvent):void
      {
        FullNameOutput.text = getBusinessInfoResult.lastResult.fullName;
        EmailOutput.text =  getBusinessInfoResult.lastResult.email;
        PositionOutput.text = getBusinessInfoResult.lastResult.position;
        for (var i:uint = 0 ; i < getBusinessInfoResult.lastResult.address.length; i++){
          addressList.addItem(getBusinessInfoResult.lastResult.address[i]);
        }
        for (var j:uint = 0 ; j < getBusinessInfoResult.lastResult.building.length; j++){
          buildingList.addItem(getBusinessInfoResult.lastResult.building[j]);
        }
        for (var k:uint = 0 ; k < getBusinessInfoResult.lastResult.office.length; k++){
          officeList.addItem(getBusinessInfoResult.lastResult.office[k]);
        }
      }
    ]]>
  </fx:Script>
  <fx:Declarations>
    <s:CallResponder id="getBusinessInfoResult"/>
    <businessservice:BusinessService id="businessService"
       fault="Alert.show(event.fault.faultString + '\n' + event.fault.faultDetail)"
       showBusyCursor="true" result="Business_result(event)"/>

  </fx:Declarations>
  <s:Button x="220" y="13" label="getBusinessInformation" id="button"
     click="button_clickHandler(event)"/>
  <s:Label x="10" y="21.5" text="FullName"/>
  <s:TextInput x="73" y="11.5" id="FullNameInput"/>
  <mx:HRule x="10" y="41" width="624"/>
  <s:Label x="341" y="197" text="Address:"/>
  <s:List x="399" y="149" dataProvider="{addressList}" id="addressOutput"></s:List>
  <s:Label x="10" y="51" text="FullName: "/>
  <s:Label x="80" y="51" text="-" id="FullNameOutput"/>
  <s:Label x="10" y="80" text="Email:"/>
  <s:Label x="10" y="116" text="Position:"/>
  <s:Label x="11" y="198" text="Office:"/>
  <s:Label x="165" y="198" text="Building:"/>
  <s:List x="48" y="150" dataProvider="{officeList}" id="officeOutput"></s:List>
  <s:List x="222" y="148" dataProvider="{buildingList}" id="buildingOutput"></s:List>
  <s:Label x="80" y="80" text="-" id="EmailOutput"/>
  <s:Label x="80" y="116" text="-" id="PositionOutput"/>
</s:Application>
```

Listing A.6 – IHM associée à l'opération `getBusinessInfo`

A.2 Schémas des compositions fonctionelles

A.2.1 $1^{ère}$ composition

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://xml.netbeans.org/schema/TestComposition"
    xmlns:tns="http://xml.netbeans.org/schema/TestComposition"
    elementFormDefault="qualified">
    <xsd:complexType name="TestCompositionInput">
        <xsd:sequence>
            <xsd:element name="CardId" type="xsd:long"/>
            <xsd:element name="FullName" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="TestCompositionOutput">
        <xsd:sequence>
            <xsd:element name="FullName" type="xsd:string"/>
            <xsd:element name="Email" type="xsd:string"/>
            <xsd:element name="InsuranceNumber" type="xsd:long"/>
        </xsd:sequence>
    </xsd:complexType>
```

```
    <xsd:element name="TestCompositionRequest" type="tns:TestCompositionInput"/>
    <xsd:element name="TestCompositionResponse" type="tns:TestCompositionOutput"/>
</xsd:schema>
```

Listing A.7 – XSD de la $1^{ème}$ composition

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="TestComposition"
   targetNamespace="http://j2ee.netbeans.org/wsdl/TestComposition"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition"
   xmlns:ns1="http://xml.netbeans.org/schema/TestComposition"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:plink="http://docs.oasis-open.org/wsbpel/2.0/plnktype">
  <types>
    <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/TestComposition"
       xmlns:ns1="http://xml.netbeans.org/schema/TestComposition">
      <xsd:import namespace="http://xml.netbeans.org/schema/TestComposition"
         schemaLocation="TestComposition.xsd"/>
    </xsd:schema>
  </types>
  <message name="TestCompositionRequest">
    <part name="input" element="ns1:TestCompositionRequest"/>
  </message>
  <message name="TestCompositionResponse">
    <part name="output" element="ns1:TestCompositionResponse"/>
  </message>
  <portType name="TestCompositionPortType_TestComposition">
    <operation name="TestCompositionOperation">
      <input name="input1" message="tns:TestCompositionRequest"/>
      <output name="output1" message="tns:TestCompositionResponse"/>
    </operation>
  </portType>
  <binding name="TestCompositionPortType_TestCompositionBinding"
     type="tns:TestCompositionPortType_TestComposition">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="TestCompositionOperation">
      <soap:operation/>
      <input name="input1">
        <soap:body use="literal"/>
      </input>
      <output name="output1">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="TestCompositionService">
    <port name="TestComposition_Port"
       binding="tns:TestCompositionPortType_TestCompositionBinding">
      <soap:address location="http://localhost:${HttpDefaultPort}/TestComposition/
                              TestComposition/TestComposition_Port"/>
    </port>
  </service>
  <plink:partnerLinkType name="TestCompositionPartnerLink">
    <plink:role name="TestCompositionPortType_TestComposition"
       portType="tns:TestCompositionPortType_TestComposition"/>
  </plink:partnerLinkType>
</definitions>
```

Listing A.8 – WSDL de la $1^{ème}$ composition

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="TestComposition"
   targetNamespace="http://enterprise.netbeans.org/bpel/TestComposition/TestComposition"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sxt="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Trace"
   xmlns:sxed="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Editor"
   xmlns:tns="http://enterprise.netbeans.org/bpel/TestComposition/TestComposition"
   xmlns:ns0="http://xml.netbeans.org/schema/TestComposition">
  <import namespace="http://j2ee.netbeans.org/wsdl/TestComposition"
     location="TestComposition.wsdl"
```

```
    importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://enterprise.netbeans.org/bpel/BusinessServiceWrapper"
    location="BusinessServiceWrapper.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://rainbow.i3s.unice.fr/business"
    location="BusinessService.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://enterprise.netbeans.org/bpel/SocialInsuranceServiceWrapper"
    location="SocialInsuranceServiceWrapper.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://rainbow.i3s.unice.fr/socialinusrance"
    location="SocialInsuranceService.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/"/>
  <partnerLinks>
    <partnerLink name="BusinessPartnerLink"
      xmlns:tns="http://enterprise.netbeans.org/bpel/BusinessServiceWrapper"
      partnerLinkType="tns:businessLinkType" partnerRole="businessRole"/>
    <partnerLink name="SocialInsurancePartnerLink"
      xmlns:tns="http://enterprise.netbeans.org/bpel/SocialInsuranceServiceWrapper"
      partnerLinkType="tns:socialinsuranceLinkType"
      partnerRole="socialinsuranceRole"/>
    <partnerLink name="TestCompoPartnerLink"
      xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition"
      partnerLinkType="tns:TestCompositionPartnerLink"
      myRole="TestCompositionPortType_TestComposition"/>
  </partnerLinks>
  <variables>
    <variable name="GetByCardOut"
      xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
      messageType="tns:getByCardResponse"/>
    <variable name="GetByCardIn"
      xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
      messageType="tns:getByCard"/>
    <variable name="GetBusinessInfoOut"
      xmlns:tns="http://rainbow.i3s.unice.fr/business"
      messageType="tns:getBusinessInfoResponse"/>
    <variable name="GetBusinessInfoIn"
      xmlns:tns="http://rainbow.i3s.unice.fr/business"
      messageType="tns:getBusinessInfo"/>
    <variable name="TestCompositionOperationOut"
      xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition"
      messageType="tns:TestCompositionResponse"/>
    <variable name="TestCompositionOperationIn"
      xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition"
      messageType="tns:TestCompositionRequest"/>
  </variables>
  <sequence>
    <receive name="Receive1" createInstance="yes"
      partnerLink="TestCompoPartnerLink" operation="TestCompositionOperation"
      xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition"
      portType="tns:TestCompositionPortType_TestComposition"
      variable="TestCompositionOperationIn"/>
    <assign name="Assign1">
      <copy>
        <from>$TestCompositionOperationIn.input/ns0:CardId</from>
        <to>$GetByCardIn.parameters/arg0</to>
      </copy>
      <copy>
        <from>$TestCompositionOperationIn.input/ns0:FullName</from>
        <to>$GetBusinessInfoIn.parameters/arg0</to>
      </copy>
    </assign>
    <flow name="Flow1">
      <invoke name="Invoke1" partnerLink="BusinessPartnerLink"
        operation="getBusinessInfo" xmlns:tns="http://rainbow.i3s.unice.fr/business"
        portType="tns:business" inputVariable="GetBusinessInfoIn"
        outputVariable="GetBusinessInfoOut"/>
      <invoke name="Invoke2" partnerLink="SocialInsurancePartnerLink"
        operation="getByCard" xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
        portType="tns:socialinsurance" inputVariable="GetByCardIn"
        outputVariable="GetByCardOut"/>
    </flow>
    <assign name="Assign2">
      <copy>
        <from>$GetBusinessInfoOut.parameters/return/fullName</from>
        <to>$TestCompositionOperationOut.output/ns0:FullName</to>
```

```
      </copy>
      <copy>
        <from>$GetBusinessInfoOut.parameters/return/email</from>
        <to>$TestCompositionOperationOut.output/ns0:Email</to>
      </copy>
      <copy>
        <from>$GetByCardOut.parameters/return/insuranceNumber</from>
        <to>$TestCompositionOperationOut.output/ns0:InsuranceNumber</to>
      </copy>
    </assign>
    <reply name="Reply1" partnerLink="TestCompoPartnerLink" operation="TestCompositionOperation"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition"
       portType="tns:TestCompositionPortType_TestComposition"
       variable="TestCompositionOperationOut"/>
  </sequence>
</process>
```

Listing A.9 – BPEL de la $1^{ème}$ composition

A.2.2 2^{nde} composition

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://xml.netbeans.org/schema/TestComposition2"
    xmlns:tns="http://xml.netbeans.org/schema/TestComposition2"
    elementFormDefault="qualified">
    <xsd:complexType name="TestCompositionInput">
        <xsd:sequence>
            <xsd:element name="CardID" type="xsd:long"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="TestCompositionOutput">
        <xsd:sequence>
            <xsd:element name="FullName" type="xsd:string"/>
            <xsd:element name="Email" type="xsd:string"/>
            <xsd:element name="PersonalAddress" type="xsd:string"/>
            <xsd:element name="BirthDay" type="xsd:dateTime"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="TestComposition2Resquest" type="tns:TestCompositionInput"/>
    <xsd:element name="TestComposition2Response" type="tns:TestCompositionOutput"/>
</xsd:schema>
```

Listing A.10 – XSD de la 2^{nde} composition

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="TestComposition2"
     targetNamespace="http://j2ee.netbeans.org/wsdl/TestComposition2"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition2"
     xmlns:ns1="http://xml.netbeans.org/schema/TestComposition2"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:plink="http://docs.oasis-open.org/wsbpel/2.0/plnktype">
    <types>
      <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/TestComposition2"
         xmlns:ns1="http://xml.netbeans.org/schema/TestComposition2">
        <xsd:import namespace="http://xml.netbeans.org/schema/TestComposition2"
           schemaLocation="TestComposition2.xsd"/>
        </xsd:schema>
    </types>
    <message name="TestComposition2Request">
        <part name="input" element="ns1:TestComposition2Resquest"/>
    </message>
    <message name="TestComposition2Response">
        <part name="output" element="ns1:TestComposition2Response"/>
    </message>
    <portType name="TestComposition2PortType_TestComposition">
        <operation name="TestComposition2Operation">
```

```
            <input name="input1" message="tns:TestComposition2Request"/>
            <output name="output1" message="tns:TestComposition2Response"/>
        </operation>
    </portType>
    <binding name="TestComposition2PortType_TestCompositionBinding"
       type="tns:TestComposition2PortType_TestComposition">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="TestComposition2Operation">
        <soap:operation/>
        <input name="input1">
           <soap:body use="literal"/>
        </input>
        <output name="output1">
           <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
    <service name="TestComposition2Service">
      <port name="TestComposition2Port_TestComposition"
         binding="tns:TestComposition2PortType_TestCompositionBinding">
        <soap:address location="http://localhost:${HttpDefaultPort}/TestComposition2/
                                TestComposition2/TestComposition2Port_TestComposition"/>
      </port>
    </service>
    <plink:partnerLinkType name="TestComposition2_TestComposition">
      <plink:role name="TestComposition2PortTyleRole_TestComposition"
         portType="tns:TestComposition2PortType_TestComposition"/>
    </plink:partnerLinkType>
</definitions>
```

Listing A.11 – WSDL de la 2nde composition

```
<?xml version="1.0" encoding="UTF-8"?>
<process
   name="TestComposition2"
   targetNamespace="http://enterprise.netbeans.org/bpel/TestComposition2/TestComposition2"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sxt="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Trace"
   xmlns:sxed="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Editor"
   xmlns:tns="http://enterprise.netbeans.org/bpel/TestComposition2/TestComposition2"
   xmlns:ns0="http://xml.netbeans.org/schema/TestComposition2">
  <import namespace="http://j2ee.netbeans.org/wsdl/TestComposition2"
     location="TestComposition2.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://enterprise.netbeans.org/bpel/BusinessServiceWrapper"
     location="BusinessServiceWrapper.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://rainbow.i3s.unice.fr/business"
     location="BusinessService.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://enterprise.netbeans.org/bpel/SocialInsuranceServiceWrapper"
     location="SocialInsuranceServiceWrapper.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://rainbow.i3s.unice.fr/socialinusrance"
     location="SocialInsuranceService.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <partnerLinks>
    <partnerLink name="BusinessPartnerLink"
       xmlns:tns="http://enterprise.netbeans.org/bpel/BusinessServiceWrapper"
       partnerLinkType="tns:businessLinkType" partnerRole="businessRole"/>
    <partnerLink name="SocialInsurancePartnerLink"
       xmlns:tns="http://enterprise.netbeans.org/bpel/SocialInsuranceServiceWrapper"
       partnerLinkType="tns:socialinsuranceLinkType"
       partnerRole="socialinsuranceRole"/>
    <partnerLink name="TestComposition2PartnerLink"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition2"
       partnerLinkType="tns:TestComposition2_TestComposition"
       myRole="TestComposition2PortTyleRole_TestComposition"/>
  </partnerLinks>
  <variables>
    <variable name="GetBusinessInfoOut" xmlns:tns="http://rainbow.i3s.unice.fr/business"
       messageType="tns:getBusinessInfoResponse"/>
    <variable name="GetBusinessInfoIn" xmlns:tns="http://rainbow.i3s.unice.fr/business"
       messageType="tns:getBusinessInfo"/>
```

```
    <variable name="GetByCardOut" xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
       messageType="tns:getByCardResponse"/>
    <variable name="GetByCardIn" xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
       messageType="tns:getByCard"/>
    <variable name="TestComposition2OperationOut"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition2"
       messageType="tns:TestComposition2Response"/>
    <variable name="TestComposition2OperationIn"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition2"
       messageType="tns:TestComposition2Request"/>
  </variables>
  <sequence>
    <receive name="Receive1" createInstance="yes"
       partnerLink="TestComposition2PartnerLink"
       operation="TestComposition2Operation"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition2"
       portType="tns:TestComposition2PortType_TestComposition"
       variable="TestComposition2OperationIn"/>
    <assign name="AssignToCallSocialInsurance">
      <copy>
        <from>$TestComposition2OperationIn.input/ns0:CardID</from>
        <to>$GetByCardIn.parameters/arg0</to>
      </copy>
    </assign>
    <invoke name="Invoke1" partnerLink="SocialInsurancePartnerLink" operation="getByCard"
       xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
       portType="tns:socialinsurance" inputVariable="GetByCardIn"
       outputVariable="GetByCardOut"/>
    <assign name="AssignToCallBusiness">
      <copy>
        <from>concat($GetByCardOut.parameters/return/firstName, ' ',
           $GetByCardOut.parameters/return/lastName)</from>
        <to>$GetBusinessInfoIn.parameters/arg0</to>
      </copy>
    </assign>
    <invoke name="Invoke2" partnerLink="BusinessPartnerLink" operation="getBusinessInfo"
       xmlns:tns="http://rainbow.i3s.unice.fr/business"
       portType="tns:business" inputVariable="GetBusinessInfoIn"
       outputVariable="GetBusinessInfoOut"/>
    <assign name="Output">
      <copy>
        <from>$GetBusinessInfoOut.parameters/return/fullName</from>
        <to>$TestComposition2OperationOut.output/ns0:FullName</to>
      </copy>
      <copy>
        <from>$GetBusinessInfoOut.parameters/return/email</from>
        <to>$TestComposition2OperationOut.output/ns0:Email</to>
      </copy>
      <copy>
        <from>$GetByCardOut.parameters/return/address</from>
        <to>$TestComposition2OperationOut.output/ns0:PersonalAddress</to>
      </copy>
      <copy>
        <from>$GetByCardOut.parameters/return/date</from>
        <to>$TestComposition2OperationOut.output/ns0:BirthDay</to>
      </copy>
    </assign>
    <reply name="Reply1" partnerLink="TestComposition2PartnerLink"
       operation="TestComposition2Operation"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition2"
       portType="tns:TestComposition2PortType_TestComposition"
       variable="TestComposition2OperationOut"/>
  </sequence>
</process>
```

Listing A.12 – BPEL de la 2nde composition

A.2.3 3ème composition

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://xml.netbeans.org/schema/TestComposition4"
```

```
    xmlns:tns="http://xml.netbeans.org/schema/TestComposition4"
    elementFormDefault="qualified">
    <xsd:complexType name="TestComposition4Input">
        <xsd:sequence>
            <xsd:element name="CardID" type="xsd:long"/>
            <xsd:element name="FullName" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="TestComposition4Output">
        <xsd:sequence>
            <xsd:element name="Addresses" type="xsd:string" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="TestComposition4Request" type="tns:TestComposition4Input"/>
    <xsd:element name="TestComposition4Response" type="tns:TestComposition4Output"/>
</xsd:schema>
```

Listing A.13 – XSD de la 3ème composition

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="TestComposition4"
   targetNamespace="http://j2ee.netbeans.org/wsdl/TestComposition4"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition4"
   xmlns:ns1="http://xml.netbeans.org/schema/TestComposition4"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:plink="http://docs.oasis-open.org/wsbpel/2.0/plnktype">
  <types>
    <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/TestComposition4"
       xmlns:ns1="http://xml.netbeans.org/schema/TestComposition4">
      <xsd:import namespace="http://xml.netbeans.org/schema/TestComposition4"
         schemaLocation="TestComposition4.xsd"/>
    </xsd:schema>
  </types>
  <message name="TestComposition4Request">
    <part name="input" element="ns1:TestComposition4Request"/>
  </message>
  <message name="TestComposition4Response">
    <part name="output" element="ns1:TestComposition4Response"/>
  </message>
  <portType name="TestComposition4PortType_TestComposition4">
    <operation name="TestComposition4Operation">
      <input name="input1" message="tns:TestComposition4Request"/>
      <output name="output1" message="tns:TestComposition4Response"/>
    </operation>
  </portType>
  <binding name="TestComposition4PortType_TestComposition4Binding"
     type="tns:TestComposition4PortType_TestComposition4">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="TestComposition4Operation">
      <soap:operation/>
      <input name="input1">
        <soap:body use="literal"/>
      </input>
      <output name="output1">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="TestComposition4Service">
    <port name="TestComposition4Port_TestComposition4"
       binding="tns:TestComposition4PortType_TestComposition4Binding">
      <soap:address location="http://localhost:${HttpDefaultPort}/TestComposition4/
                              TestComposition4/TestComposition4Port_TestComposition4"/>
    </port>
  </service>
  <plink:partnerLinkType name="TestComposition4_TestComposition4">
    <plink:role name="TestComposition4PortTypeRole_TestComposition4"
       portType="tns:TestComposition4PortType_TestComposition4"/>
  </plink:partnerLinkType>
</definitions>
```

Listing A.14 – WSDL de la 3ème composition

```
<?xml version="1.0" encoding="UTF-8"?>
<process
   name="TestComposition4"
   targetNamespace="http://enterprise.netbeans.org/bpel/TestComposition4/TestComposition4"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:sxt="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Trace"
   xmlns:sxed="http://www.sun.com/wsbpel/2.0/process/executable/SUNExtension/Editor"
   xmlns:tns="http://enterprise.netbeans.org/bpel/TestComposition4/TestComposition4"
   xmlns:ns0="http://xml.netbeans.org/schema/TestComposition4">
  <import namespace="http://j2ee.netbeans.org/wsdl/TestComposition4"
     location="TestComposition4.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://enterprise.netbeans.org/bpel/BusinessServiceWrapper"
     location="BusinessServiceWrapper.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://rainbow.i3s.unice.fr/business"
     location="BusinessService.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://enterprise.netbeans.org/bpel/SocialInsuranceServiceWrapper"
     location="SocialInsuranceServiceWrapper.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import namespace="http://rainbow.i3s.unice.fr/socialinusrance"
     location="SocialInsuranceService.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <partnerLinks>
    <partnerLink name="BusinessPartnerLink"
       xmlns:tns="http://enterprise.netbeans.org/bpel/BusinessServiceWrapper"
       partnerLinkType="tns:businessLinkType" partnerRole="businessRole"/>
    <partnerLink name="SocialInsurancePartnerLink"
       xmlns:tns="http://enterprise.netbeans.org/bpel/SocialInsuranceServiceWrapper"
       partnerLinkType="tns:socialinsuranceLinkType"
       partnerRole="socialinsuranceRole"/>
    <partnerLink name="TestComposition4PartnerLink"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition4"
       partnerLinkType="tns:TestComposition4_TestComposition4"
       myRole="TestComposition4PortTypeRole_TestComposition4"/>
  </partnerLinks>
  <variables>
    <variable name="counter" type="xsd:integer"/>
    <variable name="GetAddressesOut"
       xmlns:tns="http://rainbow.i3s.unice.fr/business"
       messageType="tns:getAddressesResponse"/>
    <variable name="GetAddressesIn"
       xmlns:tns="http://rainbow.i3s.unice.fr/business"
       messageType="tns:getAddresses"/>
    <variable name="GetByCardOut"
       xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
       messageType="tns:getByCardResponse"/>
    <variable name="GetByCardIn"
       xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
       messageType="tns:getByCard"/>
    <variable name="TestComposition4OperationOut"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition4"
       messageType="tns:TestComposition4Response"/>
    <variable name="TestComposition4OperationIn"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition4"
       messageType="tns:TestComposition4Request"/>
  </variables>
  <sequence>
    <receive name="Receive1" createInstance="yes"
       partnerLink="TestComposition4PartnerLink"
       operation="TestComposition4Operation"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition4"
       portType="tns:TestComposition4PortType_TestComposition4"
       variable="TestComposition4OperationIn"/>
    <assign name="Assign1">
      <copy>
        <from>$TestComposition4OperationIn.input/ns0:CardID</from>
        <to>$GetByCardIn.parameters/arg0</to>
      </copy>
      <copy>
        <from>$TestComposition4OperationIn.input/ns0:FullName</from>
        <to>$GetAddressesIn.parameters/arg0</to>
      </copy>
    </assign>
```

```
    <flow name="Flow1">
      <invoke name="Invoke1" partnerLink="SocialInsurancePartnerLink"
         operation="getByCard"
         xmlns:tns="http://rainbow.i3s.unice.fr/socialinusrance"
81       portType="tns:socialinsurance" inputVariable="GetByCardIn"
         outputVariable="GetByCardOut"/>
      <invoke name="Invoke2" partnerLink="BusinessPartnerLink"
         operation="getAddresses"
         xmlns:tns="http://rainbow.i3s.unice.fr/business"
86       portType="tns:business" inputVariable="GetAddressesIn"
         outputVariable="GetAddressesOut"/>
    </flow>
    <assign name="Assign2">
      <copy>
91      <from>$GetAddressesOut.parameters/return</from>
        <to>$TestComposition4OperationOut.output/ns0:Addresses</to>
      </copy>
      <copy>
        <from>count($GetAddressesOut.parameters/return)</from>
96      <to variable="counter"/>
      </copy>
      <copy>
        <from>$GetByCardOut.parameters/return/address</from>
        <to>$TestComposition4OperationOut.output/ns0:Addresses[$counter+1]</to>
101   </copy>
    </assign>
    <reply name="Reply1" partnerLink="TestComposition4PartnerLink"
       operation="TestComposition4Operation"
       xmlns:tns="http://j2ee.netbeans.org/wsdl/TestComposition4"
106    portType="tns:TestComposition4PortType_TestComposition4"
       variable="TestComposition4OperationOut"/>
  </sequence>
</process>
```

Listing A.15 – BPEL de la $3^{ème}$ composition

B
Moteur de composition

Cette annexe présente le moteur de composition qui a été réalisé et qui permet d'obtenir la composition d'IHM à partir de la composition fonctionnelle. Ce moteur a été réalisé en `Prolog` et est au cœur du processus de composition. Les sections ci-après présentent les différentes étapes du moteur de composition. Avant de commencer la section B.1 présente la représentation `Prolog` des éléments manipulés par la suite dans les différentes étapes. Elle présente ainsi la représentation des applications et de la composition fonctionnelle. L'ensemble des sections suivantes présente les différentes étapes du processus de composition. La section B.2 présente la première étape du processus de composition qui consiste à déterminer quels sont les éléments des composants fonctionnels impliqués dans la composition qui doivent être présentés. Cette première étape va permettre de déterminer quels sont les éléments graphiques qui doivent être présentés. La section B.3 présente la détermination des points de fusion puis des points de conflits. Elle présente également comment certains conflits peuvent être résolus de manière automatique. La section B.4 présente la résolution des conflits par le développeur. Cette étape reste optionnelle et dépend du résultat de la seconde étape. Enfin la section B.5 présente la création des composants d'IHM qui correspondent à la composition fonctionnelle.

B.1 Pré-requis : représentation `Prolog` des éléments nécessaires à la réalisation de la composition

L'ensemble des connaissances nécessaire à la réalisation de la composition sont représenter sous forme da fait `Prolog`. Ce qui suit présente la représentation de chacun des éléments manipulés au sein du moteur de composition avec pour commencer la représentation du composant fonctionnel, puis du composant d'IHM et pour finir d'une application. Et ensuite la représentation de la composition fonctionnelle.

B.1.1 Représentation du composant fonctionnel

Le composant fonctionnel est représenté par un ensemble de trois type de faits différents. Les faits qui permettent de décrire le composant fonctionnel sont :

– *component_fc* qui représente le composant fonctionnel avec l'ensemble des ports associés (de données et d'action) ;
– *port_definition* qui définit les ports de données avec les informations telles que le type, la cardinalité, le nom ;
– *event_data_association* qui définit les ports d'action et qui les associent au port de données.

Pour illustrer ces trois types de faits, on reprend l'exemple du composant fonctionnel qui correspond au *Web Service BusinessService*. Ce *Web Service* comporte deux opérations, chacune prend un paramètre d'entrée. La première opération retourne six éléments et la seconde un seul. Le listing B.1 décrit le composant fonctionnel du *Web Service BusinessService*.

```
component_fc(businessFC,
             [fc_input1,fc_input2],
             [fc_output1,fc_output2,fc_output3,
              fc_output4,fc_output5,fc_output6,
              fc_output7],
             [fc_action1,fc_action2]).
```

Listing B.1 – Faits `Prolog` représentant le composant fonctionnel

Le listing B.2 représente la définition d'un port de données. On retrouve également la définition du composant fonctionnel comme premier élément de ce fait. Ceci permet de garder la référence sur le composant qui possède ce port. La définition qui est faite ici correspond au premier port de données en entrée. Il a pour nom : *fullName* est de type *string* et a pour cardinalité *[1,1]*.

```
port_definition(component_fc(businessFC,
                             [fc_input1,fc_input2],
                             [fc_output1,fc_output2,fc_output3,
                              fc_output4,fc_output5,fc_output6,
                              fc_output7],
                             [fc_action1,fc_action2]),
                fc_input1,'fullName',string,[1,1]).
```

Listing B.2 – Faits `Prolog` représentant la définition d'un port de données

Le listing B.3 décrit un port d'action. Ce port d'action possède également une référence sur le composant fonctionnel qui le possède. Il permet de définir le nom de l'opération ainsi que les ports de données en entrée et en sortie qui sont associés à cette action. Ce listing reprend la première opération qui possède un paramètre d'entrée et six paramètres de sortie.

```
event_data_association(component_fc(businessFC,
                                    [fc_input1,fc_input2],
                                    [fc_output1,fc_output2,fc_output3,
                                     fc_output4,fc_output5,fc_output6,
                                     fc_output7],
                                    [fc_action1,fc_action2]),
                       fc_action1,'getBusinessInfo',
                       [fc_input1],
                       [fc_output1,fc_output2,fc_output3,
                        fc_output4,fc_output5,fc_output6]).
```

Listing B.3 – Faits `Prolog` représentant la définition d'un port d'action

B.1.2 Représentation du composant d'IHM

De la même manière que pour le composant fonctionnel, le composant d'IHM possède également trois types de faits qui permettent de le décrire. Ces faits sont identiques, seul le nombre d'éléments varie lors de la description des ports de données. En effet, ceux-ci prennent une valeur de plus dans le cas des entrées puisque l'on a également la caractéristique de sélection qu'il est possible d'effectuer.

L'illustration des différents types de faits est effectuée sur l'IHM qui permet d'obtenir l'ensemble des informations pour un employé. Elle comprend une entrée, un bouton et six sorties.

Le listing B.4 représente le composant d'IHM avec l'ensemble des ports qu'il possède.

```
component_aui(getBusinessInfoUI,
              [ui_input1],
              [ui_output1,ui_output2,ui_output3,
               ui_output4,ui_output5,ui_output6],
              [ui_action1]).
```

Listing B.4 – Faits `Prolog` représentant le composant d'IHM

Le listing B.5 représente la définition du port de données en entrée du composant d'IHM. Ce port d'entrée possède comme nom *fullName*, comme type *string*, comme cardinalité *1* (ce qui signifie que c'est un élément unaire) et comme sélection "-" puisqu'il n'est pas possible d'avoir de notion de sélection sur un élément unaire. Les valeurs possibles prises par la sélection sont identiques à celles décrites dans la formalisation c'est-à-dire : *NONE, SINGLE, MULTIPLE*.

```
port_definition(component_aui(getBusinessInfoUI,
                              [ui_input1],
                              [ui_output1,ui_output2,ui_output3,
                               ui_output4,ui_output5,ui_output6],
                              [ui_action1]),
                ui_input1,'fullName',string,1,-).
```

Listing B.5 – Faits `Prolog` représentant le port d'entrée du composant d'IHM

Le listing B.6 décrit un port de sortie du composant d'IHM. Celui-ci correspond au champ qui reçoit les informations concernant les différents adresses d'un employé. On voit ainsi que la cardinalité vaut *n* car il doit être possible d'afficher plusieurs adresses en même temps.

```
port_definition(component_aui(getBusinessInfoUI,
                              [ui_input1],
                              [ui_output1,ui_output2,ui_output3,
                               ui_output4,ui_output5,ui_output6],
                              [ui_action1]),
                ui_output1,'address',string,n).
```

Listing B.6 – Faits `Prolog` représentant un port de sortie du composant d'IHM

Le listing B.7 représente le bouton qui permet d'appeler l'opération du *Web Service*. De la même manière que pour le composant fonctionnel, cette représentation fait le lien entre un événement et des entrées/sorties. Le nom associé au bouton est *getBusinessInfo*.

```
event_data_association(component_aui(getBusinessInfoUI,
                                     [ui_input1],
                                     [ui_output1,ui_output2,ui_output3,
                                      ui_output4,ui_output5,ui_output6],
                                     [ui_action1]),
                       ui_action1,'getBusinessInfo',
                       [ui_input1],
                       [ui_output1,ui_output2,ui_output3,
                       ui_output4,ui_output5,ui_output6]).
```

Listing B.7 – Faits `Prolog` représentant le port d'événement du composant d'IHM

B.1.3 Représentation de l'application

Une application est représentée par trois types de faits différents qui permettent de décrire :
- l'association entre un composant fonctionnel et des composants d'IHM (*association*) ;
- les liens de données entre composant fonctionnel et composant d'IHM (*data_link*) ;
- les liens d'événement entre un composant d'IHM et le composant fonctionnel (*event_link*).

Le listing B.8 représente l'association entre un composant fonctionnel et ses composants d'IHM. On retrouve la référence vers le composant fonctionnel et une liste de références vers les composants d'IHM.

```
association(component_fc(businessFC,
                         [fc_input1,fc_input2],
                         [fc_output1,fc_output2,fc_output3,
                          fc_output4,fc_output5,fc_output6,
                          fc_output7],
                         [fc_action1,fc_action2]),
            [component_aui(getBusinessInfoUI,
                           [ui_input1],
                           [ui_output1,ui_output2,ui_output3,
                            ui_output4,ui_output5,ui_output6],
                           [ui_action1]),
             component_aui(getAddressesUI,[ui_input1],
                                          [ui_output1],
                                          [ui_action1])]).
```

Listing B.8 – Faits `Prolog` représentant l'association entre composant fonctionnel et composant d'IHM

Le listing B.9 représente un lien de données entre deux entrées. La description des liens de données entre deux entrées ou entre deux sorties est identique, seul l'ordre de déclaration des composants permet de savoir si c'est un lien entre deux entrées ou entre deux sorties. Dans le cas du listing B.9, le premier composant fait référence à un composant d'IHM et le second composant au composant fonctionnel. Ensuite, on trouve une liste de couple qui correspond aux liens. Dans ce cas, on a qu'un seul lien entre deux entrées.

```
data_link(component_aui(getBusinessInfoUI,
                        [ui_input1],
                        [ui_output1,ui_output2,ui_output3,
                         ui_output4,ui_output5,ui_output6],
                        [ui_action1]),
          component_fc(businessFC,
                       [fc_input1,fc_input2],
                       [fc_output1,fc_output2,fc_output3,
                        fc_output4,fc_output5,fc_output6,
                        fc_output7],
                       [fc_action1,fc_action2]),
          [[ui_input1,fc_input1]]).
```

Listing B.9 – Faits `Prolog` représentant un lien de données entre deux entrées

Le listing B.10 décrit des liens de données entre deux sorties. Ce sont des liens entre sorties puisque le premier composant est un composant fonctionnel et le second un composant d'IHM. On retrouve ensuite une liste de couple qui permet de décrire les liens. Ici, on a six liens entre le composant fonctionnel et le composant d'IHM.

```
data_link(component_fc(businessFC,
                       [fc_input1,fc_input2],
                       [fc_output1,fc_output2,fc_output3,
                        fc_output4,fc_output5,fc_output6,
                        fc_output7],
                       [fc_action1,fc_action2]),
          component_aui(getBusinessInfoUI,
                        [ui_input1],
                        [ui_output1,ui_output2,ui_output3,
                         ui_output4,ui_output5,ui_output6],
                        [ui_action1]),
          [[fc_output1,ui_output1],[fc_output2,ui_output2],[fc_output3,ui_output3],
           [fc_output4,ui_output4],[fc_output5,ui_output5],[fc_output6,ui_output6]]).
```

Listing B.10 – Faits `Prolog` représentant un lien de données entre des sorties

Le listing B.11 décrit un lien d'événement. Ce lien d'événement se fait entre un composant d'IHM et le composant fonctionnel. De la même manière que précédemment pour les liens de données on retrouve une liste de couple qui permet de faire le lien entre un événement du composant d'IHM vers une action du composant fonctionnel.

```
event_link(component_aui(getBusinessInfoUI,
                         [ui_input1],
```

```
                        [ui_output1,ui_output2,ui_output3,
                         ui_output4,ui_output5,ui_output6],
                        [ui_action1]),
          component_fc(businessFC,
                       [fc_input1,fc_input2],
                       [fc_output1,fc_output2,fc_output3,
                        fc_output4,fc_output5,fc_output6,
                        fc_output7],
                       [fc_action1,fc_action2]),
          [[ui_action1,fc_action1]]).
```

Listing B.11 – Faits Prolog représentant un lien d'événement

B.1.4 Représentation de la composition fonctionnelle

La composition fonctionnelle est décrite au travers de cinq types de faits. Ces faits sont :

- *composite_fc* qui décrit le composant composite avec l'ensemble des ports d'entrée et de sortie et d'action comme pour un composant fonctionnel. Il possède en plus par rapport au composant fonctionnel une liste des sous-composants fonctionnels qui sont impliqués dans la composition ;
- *port_definition* qui définit de la même manière que pour les ports de données d'un composant fonctionnels les caractéristiques d'un port de donnée du composant composite ;
- *event_data_association* qui permet de faire le liens entre une action et les paramètres d'entrée et de sorties ;
- *data_link* qui permet de décrire des liens de données au sein de la composition fonctionnelle et qui permet de relier des ports du composant composite à ses sous-composants ou de lier des sous-composants entre eux ;
- *event_link* qui décrit les liens entre deux actions soit entre le composant composite et un de ses sous-composants soit entre sous-composants.

Le listing B.12 représente la définition de la composition fonctionnelle décrite en section 2.3.1. Cette composition fonctionnelle fait une sélection sur les sorties présentées pour n'en conserver qu'un certain nombre. Ce composant fonctionnel utilise deux sous-composants qui sont les composants *businessFC* et *socialInsuranceFC*. Ce composant composite possède deux ports d'entrée et trois ports de sorties et qu'une seule action.

```
composite_fc(firstComposition,
             [component_fc(businessFC,
                           [fc_input1,fc_input2],
                           [fc_output1,fc_output2,fc_output3,
                            fc_output4,fc_output5,fc_output6,
                            fc_output7],
                           [fc_action1,fc_action2]),
              component_fc(socialInsuranceFC,
                           [fc_input1,fc_input2,fc_input3],
                           [fc_output1,fc_output2,fc_output3,
                            fc_output4,fc_output5,fc_output6,
                            fc_output7,fc_output8,fc_output9,
                            fc_output10,fc_output11,fc_output12,
                            fc_output13,fc_output14],
                           [fc_action1,fc_action2])],
              [fc_input1,fc_input2],
              [fc_output1,fc_output2,fc_output3],
              [fc_action1]).
```

Listing B.12 – Faits Prolog représentant le composant composite

Le listing B.13 décrit la définition du premier port d'entrée du composant composite. Ce port permet de fournir les informations concernants la carte vitale d'un assuré. Il est définit par un nom *cardID*, un type *long* et une cardinalité *[1,1]*.

```
port_definition(composite_fc(firstComposition,
                             [component_fc(businessFC,
                                           [fc_input1,fc_input2],
```

```
                                        [fc_output1,fc_output2,fc_output3,
                                         fc_output4,fc_output5,fc_output6,
                                         fc_output7],
                                        [fc_action1,fc_action2]),
                                component_fc(socialInsuranceFC,
                                        [fc_input1,fc_input2,fc_input3],
                                        [fc_output1,fc_output2,fc_output3,
                                         fc_output4,fc_output5,fc_output6,
                                         fc_output7,fc_output8,fc_output9,
                                         fc_output10,fc_output11,fc_output12,
                                         fc_output13,fc_output14],
                                        [fc_action1,fc_action2])],
                                [fc_input1,fc_input2],
                                [fc_output1,fc_output2,fc_output3],
                                [fc_action1]),
                        fc_input1,'cardID',long,[1,1]).
```

Listing B.13 – Faits `Prolog` représentant la définition d'un port de données

Le listing B.14 définit l'action présente au sein du composant composite et qui correspond à l'opération fournie par la composition fonctionnelle. Celle-ci se nomme *getInfo* et est associée à deux ports d'entrée et trois ports de sorties.

```
event_data_association(composite_fc(firstComposition,
                                [component_fc(businessFC,
                                        [fc_input1,fc_input2],
                                        [fc_output1,fc_output2,fc_output3,
                                         fc_output4,fc_output5,fc_output6,
                                         fc_output7],
                                        [fc_action1,fc_action2]),
                                 component_fc(socialInsuranceFC,
                                        [fc_input1,fc_input2,fc_input3],
                                        [fc_output1,fc_output2,fc_output3,
                                         fc_output4,fc_output5,fc_output6,
                                         fc_output7,fc_output8,fc_output9,
                                         fc_output10,fc_output11,fc_output12,
                                         fc_output13,fc_output14],
                                        [fc_action1,fc_action2])],
                                 [fc_input1,fc_input2],
                                 [fc_output1,fc_output2,fc_output3],
                                 [fc_action1]),
                        fc_action1,'getInfo',
                        [fc_input1,fc_input2],
                        [fc_output1,fc_output2,fc_output3]).
```

Listing B.14 – Faits `Prolog` représentant la définition du port d'action

Le listing B.15 définit les liens de données entre deux entrées. Ces liens se font entre le composant composite et un de ses sous-composants. Ici, les liens sont faits vers le composant *businessFC*. De la même manière que pour une application, ici on a une liste de couple qui permet de décrire les liens. Un seul lien est effectué entre le composant composite et le composant *businessFC*.

```
data_link(composite_fc(firstComposition,
                        [component_fc(businessFC,
                                        [fc_input1,fc_input2],
                                        [fc_output1,fc_output2,fc_output3,
                                         fc_output4,fc_output5,fc_output6,
                                         fc_output7],
                                        [fc_action1,fc_action2]),
                         component_fc(socialInsuranceFC,
                                        [fc_input1,fc_input2,fc_input3],
                                        [fc_output1,fc_output2,fc_output3,
                                         fc_output4,fc_output5,fc_output6,
                                         fc_output7,fc_output8,fc_output9,
                                         fc_output10,fc_output11,fc_output12,
                                         fc_output13,fc_output14],
                                        [fc_action1,fc_action2])],
                         [fc_input1,fc_input2],
                         [fc_output1,fc_output2,fc_output3],
                         [fc_action1]),
          component_fc(businessFC,
                        [fc_input1,fc_input2],
                        [fc_output1,fc_output2,fc_output3,
```

```
                        fc_output4,fc_output5,fc_output6,
                        fc_output7],
                       [fc_action1,fc_action2]),
          [[fc_input2,fc_input1]]).
```

Listing B.15 – Faits Prolog représentant la définition des liens entre entrées du composant composite et un des sous-composants

Le listing B.16 définit des liens entre sorties. Ici entre des sorties du composant *businessFC* et le composant composite. Ici, il y a deux liens qui sont décrits.

```
data_link(component_fc(businessFC,
                       [fc_input1,fc_input2],
                       [fc_output1,fc_output2,fc_output3,
                        fc_output4,fc_output5,fc_output6,
                        fc_output7],
                       [fc_action1,fc_action2]),
          composite_fc(firstComposition,
                       [component_fc(businessFC,
                                     [fc_input1,fc_input2],
                                     [fc_output1,fc_output2,fc_output3,
                                      fc_output4,fc_output5,fc_output6,
                                      fc_output7],
                                     [fc_action1,fc_action2]),
                        component_fc(socialInsuranceFC,
                                     [fc_input1,fc_input2,fc_input3],
                                     [fc_output1,fc_output2,fc_output3,
                                      fc_output4,fc_output5,fc_output6,
                                      fc_output7,fc_output8,fc_output9,
                                      fc_output10,fc_output11,fc_output12,
                                      fc_output13,fc_output14],
                                     [fc_action1,fc_action2])],
                        [fc_input1,fc_input2],
                        [fc_output1,fc_output2,fc_output3],
                        [fc_action1]),
          [[fc_output4,fc_output1],[fc_output3,fc_output2]]).
```

Listing B.16 – Faits Prolog représentant la définition des liens entre sorties du composant composite et un des sous-composants

Le listing B.17 décrit un lien d'action entre le composant composite et le sous-composant *businessFC*. Ici, un seul lien est décrit.

```
event_link(composite_fc(firstComposition,
                        [component_fc(businessFC,
                                      [fc_input1,fc_input2],
                                      [fc_output1,fc_output2,fc_output3,
                                       fc_output4,fc_output5,fc_output6,
                                       fc_output7],
                                      [fc_action1,fc_action2]),
                         component_fc(socialInsuranceFC,
                                      [fc_input1,fc_input2,fc_input3],
                                      [fc_output1,fc_output2,fc_output3,
                                       fc_output4,fc_output5,fc_output6,
                                       fc_output7,fc_output8,fc_output9,
                                       fc_output10,fc_output11,fc_output12,
                                       fc_output13,fc_output14],
                                      [fc_action1,fc_action2])],
                        [fc_input1,fc_input2],
                        [fc_output1,fc_output2,fc_output3],
                        [fc_action1]),
           component_fc(businessFC,
                        [fc_input1,fc_input2],
                        [fc_output1,fc_output2,fc_output3,
                         fc_output4,fc_output5,fc_output6,
                         fc_output7],
                        [fc_action1,fc_action2]),
           [[fc_action1,fc_action1]]).
```

Listing B.17 – Faits Prolog représentant la définition des liens entre actions du composant composite et un des sous-composants

Ceci conclut la description de l'ensemble des éléments qui sont nécessaires à la réalisation de la composition. A partir de ceux-ci, il est possible de déduire les éléments d'IHM à conserver et de créer les composants d'IHM associés à la composition fonctionnelle (donc au composant composite).

B.2 Première étape : détermination des éléments à présenter

La première étape consiste à déterminer les ports des sous-composants qui sont utilisés par le composant composite et dont on souhaite avoir la représentation graphique correspondante. Pour ce faire, il a été nécessaire de mettre en place deux prédicats qui permettent d'obtenir les premiers éléments d'une liste de couple et les seconds éléments d'une liste de couple (cf. listing B.18).

```
/**
  *
  * PRED: get_second_element( +ListSource, -List)
  *
  * Create a list of that take the second element of each element of the
  * source list (which has the following representation [[a,b],[c,d],...]
  *
  * Arg 2 is the resulting list (e.g. [b,d,...])
  *
  */
get_second_element([],[]).
get_second_element([[_A,B]|Queue],[B|NewQueue]):-
        get_second_element(Queue,NewQueue),!.
get_second_element([_A,B],[B]).

/**
  *
  * PRED: get_first_element( +ListSource, -List)
  *
  * Create a list of that take the first element of each element of the
  * source list (which has the following representation [[a,b],[c,d],...]
  *
  * Arg 2 is the resulting list (e.g. [a,c,...])
  *
  */
get_first_element([],[]).
get_first_element([[A,_B]|Queue],[A|NewQueue]):-
        get_first_element(Queue,NewQueue),!.
get_first_element([A,_B],[A]).
```

Listing B.18 – Prédicats permettants d'obtenir les premiers ou les seconds éléments d'une liste de couple

Pour chacun des types ports présents, un prédicat est défini.Le résultat obtenu est une liste qui contient pour l'ensemble des sous-composants la liste des ports présentés. La liste ainsi obtenue est de la forme : $[[SubComponent, [PresentedPorts]], [SubComponent, [PresentedPorts]], ...]$

Le listing B.19 présente le prédicat : *in_presented* qui permet d'obtenir la liste des entrées présentées. La calcul de cette liste se fait en utilisant la liste des sous-composants présents au sein d'un composant composite. Pour chacun de ces sous-composants, on prend la liste des liens de données qui existent (dans ce cas, ce sont des liens entre entrées). Enfin de cette liste de couple, on ne conserve dans ce cas que les seconds éléments. Il en est de même pour obtenir les actions présentées. Par contre dans le cas des sorties, on ne conserve que les premiers éléments.

```
/**
  *
  * PRED: in_presented( +CompositeFC, +SubComponents, -List)
  *
  * Create a list of presented in ports that we have to find their UI
  * corresponding ports
  *
  * Arg 3 is the result of the research of presented ports which are a list of
  * [[SubComponent,[PresentedPorts]],...]
```

```
  *
  */
in_presented(_ComposeFC,[],[]).
in_presented(ComposeFC,[SubComponent|ListSubComponents],
             [[SubComponent,PortIn]|OtherPresented]):-
        data_link(composite_fc(ComposeFC,_,_,_,_),SubComponent,ListLink),
        get_second_element(ListLink,PortIn),
        in_presented(ComposeFC,ListSubComponents,OtherPresented),!.
in_presented(ComposeFC,SubComponent,[[SubComponent,PortIn]]):-
        data_link(composite_fc(ComposeFC,_,_,_,_),SubComponent,ListLink),
        get_second_element(ListLink,PortIn).
```

Listing B.19 – Prédicat pour obtenir la liste des entrées présentées

Les listing B.20 et B.21 correspondent aux prédicats qui permettent de créer la liste des sorties présentées et des actions présentées. On remarque également que pour obtenir la liste des actions présentées on utilise les *event_link* à la place des *data_link*. Sinon le mécanisme utilisé est identique.

```
out_presented(_ComposeFC,[],[]).
out_presented(ComposeFC,[SubComponent|ListSubComponents],
              [[SubComponent,PortOut]|OtherPresented]):-
        data_link(SubComponent,composite_fc(ComposeFC,_,_,_,_),ListLink),
        get_first_element(ListLink,PortOut),
        out_presented(ComposeFC,ListSubComponents,OtherPresented),!.
out_presented(ComposeFC,SubComponent,[[SubComponent,PortOut]]):-
        data_link(SubComponent,composite_fc(ComposeFC,_,_,_,_),ListLink),
        get_first_element(ListLink,PortOut).
```

Listing B.20 – Prédicat pour obtenir la liste des sorties présentées

```
action_presented(_ComposeFC,[],[]).
action_presented(ComposeFC,[SubComponent|ListSubComponents],
                 [[SubComponent,PortAction]|OtherPresented]):-
        event_link(composite_fc(ComposeFC,_,_,_,_),SubComponent,ListLink),
        get_second_element(ListLink,PortAction),
        action_presented(ComposeFC,ListSubComponents,OtherPresented),!.
action_presented(ComposeFC,SubComponent,[[SubComponent,PortAction]]):-
        event_link(composite_fc(ComposeFC,_,_,_,_),SubComponent,ListLink),
        get_second_element(ListLink,PortAction).
```

Listing B.21 – Prédicat pour obtenir la liste des actions présentées

A partir de ces listes, il est possible d'obtenir la liste des éléments d'IHM associés à chacun des éléments présentés puis par extension, de connaître pour chacun des ports du composant composite les éléments d'IHM associés. Ces listes permettent également de déterminer si un point de fusion et un point de conflit ou pas. Pour ce faire, il est nécessaire de réaliser des étapes intermédiaires qui permettent :

1. De créer une liste qui associe à chacun des ports du composant composite une liste des ports des sous-composants auquel il est lié ;
2. De créer une liste qui pour chacun des éléments présentés associe sa représentation graphique ;
3. De créer une liste qui à partir des deux listes précédentes permet de lier à un port du composant composite l'ensemble des représentations graphiques qu'il peut avoir.

B.2.1 Création de trois listes qui mettent en relation les ports du composant composite avec les ports des sous-composants

La première étape crée trois listes pour chacun des types de ports. Les listes ainsi créées ont pour forme : $[[PortIn, [[SubComponent, [Ports]]]], ...]$ (de même pour les autres types de ports). Ainsi, pour chacun des ports du composant composite, on a une association qui est faite avec pour chacun des sous-composants où il existe une liaison sur ce port la liste des ports auquel il est connecté.

Le listing B.22 présente le prédicat *create_list_by_port_in* qui permet de créer la liste précédemment décrite pour les ports d'entrée du composant composite. Ce prédicat est déroulé de manière récursive sur chacun des ports d'entrée du composant composite. Il utilise pour ce faire le nom du composant composite, la liste des ports d'entrée, la listes des sous-composants. Enfin la dernière liste correspond au résultat. Celui-ci fait appel à un autre prédicat : *create_list_port_in*.

```
/**
  *
  * PRED: create_list_by_port_in( +CompositeFC, +PortsIn, +SubComponents, -List)
  *
  * Create a list by port in of the composite component to know on which port
  * of the sub components it is connected
  *
  * Arg 4 is the result of the research of linked ports with the form
  * [[PortIn,[[SubComponent,[Ports]]]],...]
  *
  */
create_list_by_port_in(_CompositeFC,[],_SubComponents,[]).
create_list_by_port_in(CompositeFC,[PortIn|OtherPorts],SubComponents,[[PortIn,Res]|Others]):-
        create_list_port_in(CompositeFC,PortIn,SubComponents,Res),
        create_list_by_port_in(CompositeFC,OtherPorts,SubComponents,Others),!.
create_list_by_port_in(CompositeFC,[PortIn],SubComponents,[[PortIn,Res]]):-
        create_list_port_in(CompositeFC,PortIn,SubComponents,Res).
```

Listing B.22 – Prédicats permettants d'obtenir la liste qui associe à chaque port la liste des ports des sous-composants auquel il est reliés (dans le cas des entrées)

Le listing B.23 présente comment est obtenu la liste de la forme $[[SubComponent, [Ports]], ...]$ associée à chacun des ports du composant composite. Pour ce faire, il est nécessaire d'itérer sur chacun des sous-composants pour un port d'entrée donné. La première étape consiste à obtenir les liens de données qu'il y a entre le composant composite et le sous-composant traité. Dans le cas où il n'y en a pas, on passe au sous-composant suivant. Dans le cas où il y a un ou plusieurs liens, on fait appel au prédicat *create_list_in_action* (dans le cas d'une entrée ou d'une action) ou au prédicat *create_list_out* (dans le cas d'une sortie). Ce prédicat permet d'obtenir une liste de ports (dans le cas où des liens existent avec le port du composant composite qui est traité) ou bien une liste vide dans le cas où il n'y a pas de lien. C'est pourquoi un test est effectué afin de vérifier que la liste contient des ports ou non avant d'obtenir le résultat.

```
/**
  *
  * PRED: create_list_port_in( +Composite, +PortIn, +SubComponents, ?Res)
  *
  * Create a list of connected ports
  *
  * Arg 4 is the resulting list which have this representation
  * [[SubComponent,[Ports]],...]
  *
  */
create_list_port_in(_CompositeFC,_PortIn,[],[]).
create_list_port_in(CompositeFC,PortIn,[SubComponent|OtherComponents],Res):-
        data_link(composite_fc(CompositeFC,_,_,_,_),SubComponent,List),
        create_list_in_action(PortIn,List,ResultingList),
        length(ResultingList,Size),
        (Size > 0 -> append([[SubComponent,ResultingList]],Others,Res);
            append([],Others,Res)
        ),
        create_list_port_in(CompositeFC,PortIn,OtherComponents,Others),!.
create_list_port_in(CompositeFC,PortIn,[SubComponent|OtherComponents],ListRes):-
        not(data_link(composite_fc(CompositeFC,_,_,_,_),SubComponent,_List)),
        create_list_port(CompositeFC,PortIn,OtherComponents,ListRes),!.
create_list_port_in(CompositeFC,PortIn,[SubComponent],[[SubComponent,ResultingList]]):-
        data_link(composite_fc(CompositeFC,_,_,_,_),SubComponent,List),
        create_list_in_action(PortIn,List,ResultingList),
        length(ResultingList,Size),
        (Size > 0 -> append([[SubComponent,ResultingList]],[],Res);
            append([],[],Res)
        ).
create_list_port_in(CompositeFC,_PortIn,[SubComponent],[]):-
        not(data_link(composite_fc(CompositeFC,_,_,_,_),SubComponent,_List)).
```

Listing B.23 – Liste qui associe à un sous-composant l'ensemble de ses ports qui sont liés à un même port du composant composite

Le listing B.24 présente le prédicat *create_list_port_in_action* qui à partir d'un port donné du composant composite et d'une liste de liens permet de créer une liste qui contient l'ensemble des ports qui sont liés au port du composant composite (et qui sont présents dans la liste de liens traitée).

```
/**
  *
  * PRED: create_list_port_in_action(+PortInAction, +ListLink, ?Res)
  *
  * Create a list of extracted ports
  *
  * Arg 3 is the resulting list which have this representation
  * [Ports]
  *
  */
create_list_in_action(_PortInAction,[],[]).
create_list_in_action(PortInAction,[[PortInAction,InAction]|ListLink],[InAction|Queue]):-
        create_list_in_action(PortInAction,ListLink,Queue),!.
create_list_in_action(PortInAction,[[_OtherInAction,_InAction]|ListLink],List):-
        create_list_in_action(PortInAction,ListLink,List),!.
create_list_in_action(PortInAction,[PortInAction,InAction],[InAction]).
create_list_in_action(_PortInAction,[_OtherInAction,_InAction],[]).
```

Listing B.24 – Liste qui contient l'ensemble des ports liés à un port du composant composite pour un sous-composant donné

B.2.2 Création de trois listes qui associent à chacun des éléments présentés sa représentation graphique

L'objectif ici est de créer trois listes qui permettent d'associer à chacun des ports présentés sa représentation graphique. Les prédicats présentés ici s'applique sur les entrées mais il existe des prédicats équivalents pour les sorties et les actions.

Le listing B.25 présente le prédicat qui permet d'obtenir la liste finale qui est de la forme : $[[SubComponent, [[PortInFC, [UIComponent, PortInUI, Name, Type, Arity, Selection]], ...], ...], ...]$. Pour réaliser cette liste, on s'appuie sur la liste précédemment créée qui contient la liste des ports présentés pour chacun des sous-composants. On itère ainsi sur chacun des couples *[sous-composants,ports présentés]*. Pour chacun de ces couples on fait appel au prédicat *obtain_ui_information_in* qui va permettre de calculer le résultat voulu.

```
/**
  *
  * PRED: obtain_presented_ui_in( +ListSource, ?List)
  *
  * Create a list of presented ui portIn in function of subcomponent and
  * its port
  *
  * Arg 2 is the resulting list which looks like :
  * [[SubComponent,[[PortInFC,[UIComponent,PortInUI,Name,Type,Arity,Selection]],
  * ...],...],...]
  *
  */
obtain_presented_ui_in([],[]).
obtain_presented_ui_in([[SubComponents,PresentedPort]|OthersPresented],
                       [[SubComponents,Res]|OtherRes]):-
        obtain_ui_information_in(SubComponents,PresentedPort,Res),
        obtain_presented_ui_in(OthersPresented,OtherRes),!.
obtain_presented_ui_in([[SubComponents,PresentedPort]],[SubComponents,Res]) :-
        obtain_ui_information_in(SubComponents,PresentedPort,Res).
```

Listing B.25 – Prédicat qui permet d'associer à chaque port présenté d'un sous-composant, sa représentation graphique

Le listing B.26 présente le prédicat *obtain_ui_information_in*. Ce prédicat permet de créer une liste qui a pour forme : [[*PresentedPortFC*, [*UIComponent*, *PresentedPortUI*, *Name*, *Type*, *Arity*, *Selection*]], ...]. Cette représentation est propre au port d'entrée. Dans le cas des ports de sortie, il n'y a pas de *Selection* et dans le cas d'action, il n'y a que le nom (*Name*). Ce prédicat itère sur l'ensemble des ports présentés et la création du résultat se fait en réutilisant le port présenté auquel on associe la liste des informations graphiques. Pour ce faire, on recherche les informations graphiques en utilisant le prédicat *obtain_ui_in_information* qui itère sur chacun des composants graphiques auquel le sous-composant fonctionnel est associé.

```
/**
  *
  * PRED: obtain_ui_information_in( +SubComponent, +ListPresentedPort, ?List)
  *
  * Create a list of presented ui portIn definition for all presented port of
  * a subcomponent
  *
  * Arg 3 is the resulting list which looks like :
  * [[PresentedPortFC,[UIComponent,PresentedPortUI,Name,Type,Arity,Selection]],
  *  ...]
  *
  */
obtain_ui_information_in(_SubComponent,[],[]).
obtain_ui_information_in(SubComponent,[PresentedPort|OthersPresentedPorts],
                         [[PresentedPort,ResultInfo]|Other]):-
        association(SubComponent,ListUIComponent),!,
        obtain_ui_in_information(SubComponent,PresentedPort,ListUIComponent,ResultInfo),
        obtain_ui_information_in(SubComponent,OthersPresentedPorts,Other),!.
```

Listing B.26 – Prédicat qui permet d'associer à un port sa représentation graphique

Le listing B.27 présente le prédicat *obtain_ui_in_information* et également le prédicat *looking_for_ui_in_port_definition* auquel il fait appel. Le prédicat *obtain_ui_in_information* itère sur chacun des composants d'IHM auquel est associé le composant fonctionnel. Dès qu'une représentation graphique associé au port présenté est trouvé, l'itération s'achève. Le prédicat *looking_for_ui_in_port_definition* permet d'aller obtenir les informations de représentation graphique en se basant sur les liens de données qui existent entre le composant fonctionnel et le composant d'IHM. Pour ce faire, il utilise le prédicat *get_ui_in_information*.

```
/**
  *
  * PRED: obtain_ui_in_information( +SubComponent, +PresentedPort, +ListUIComponent, ?List)
  *
  * Create a list of presented ui portIn definition for all presented port of
  * a subcomponent
  *
  * Arg 4 is the resulting list which looks like :
  * [UIComponent,PresentedPortUI,Name,Type,Arity,Selection]
  *
  */
obtain_ui_in_information(SubComponent,PresentedPort,[UIComponent|_OtherComponent],ResultInfo):-
        looking_for_ui_in_port_definition(SubComponent,PresentedPort,UIComponent,ResultInfo),!.
obtain_ui_in_information(SubComponent,PresentedPort,[UIComponent|OtherComponent],List):-
   not(looking_for_ui_in_port_definition(SubComponent,PresentedPort,UIComponent,_ResultInfo)),
   obtain_ui_in_information(SubComponent,PresentedPort,OtherComponent,List),!.

/**
  *
  * PRED: looking_for_ui_in_port_definition( +SubComponent, +PresentedPort, +UIComponent, ?List)
  *
  * Create a list of presented ui portIn definition for all presented port of
  * a subcomponent
  *
  * Arg 4 is the resulting list which looks like :
  * [UIComponent,PresentedPortUI,Name,Type,Arity,Selection]
  *
  */
looking_for_ui_in_port_definition(SubComponent,PresentedPort,UIComponent,ResultInfo):-
        data_link(UIComponent,SubComponent,List),!,
        get_ui_in_information(PresentedPort,UIComponent,List,ResultInfo).
```

Listing B.27 – Prédicat qui permet d'obtenir la représentation graphique

Le listing B.28 présente le prédicat *get_ui_in_information* qui permet de créer une liste contenant la description graphique associé à un port. Pour ce faire, il itère sur l'ensemble des liens jusqu'à ce qu'il trouve le lien qui correspond au port fonctionnel recherché.

```
/**
  *
  * PRED: get_ui_in_information( +PresentedPort, +UIComponent, +ListLink, ?List)
  *
  * Create a list of presented ui portIn definition for all presented port of
  * a subcomponent
  *
  * Arg 4 is the resulting list which looks like :
  * [UIComponent,PresentedPortUI,Name,Type,Arity,Selection]
  *
  */
get_ui_in_information(PresentedPort,UIComponent,[[UIPort,PresentedPort]|_OtherLink],
                      [UIComponent,UIPort,Name,Type,Arity,Selection]):-
        port_definition(UIComponent,UIPort,Name,Type,Arity,Selection),!.
get_ui_in_information(PresentedPort,UIComponent,[[_UIPort,_OtherPort]|OtherLink],List):-
        get_ui_in_information(PresentedPort,UIComponent,OtherLink,List).
```

Listing B.28 – Prédicat qui extrait les informations de la représentation graphique

B.2.3 Création de trois listes qui permettent de relier les ports du composant composite à l'ensemble des représentations graphiques qu'ils peuvent avoir

La création de ces trois listes (*AssociatedFCPortInWithUI*, *AssociatedFCPortOutWithUI* et *AssociatedFCPortActionWithUI*) s'appuie sur les deux types de listes précédemment créés. L'objectif ici est d'obtenir une liste qui associe à chacun des ports du composant composite l'ensemble des représentations graphiques qu'il possède. La liste obtenue a pour forme : $[[Port, [[UIDefinition], ...]], ...]$.

Le listing B.29 présente le prédicat *get_all_associated_port_with_ui_definition* qui permet d'obtenir une telle liste. Ce prédicat prend en premier paramètre la liste qui associe à chacun des ports les ports des sous-composants auxquels il est lié (cf. listing B.22) et en second paramètre la liste qui associe à chaque port des sous-composants sa représentation graphique (cf. listing B.25). Pour ce faire, il fait appel au prédicat *all_associated_port_with_ui_definition* qui pour un port du composant composite va calculer la liste des représentations graphiques.

```
/**
  *
  * PRED: get_all_associated_port_with_ui_definition(+ListCompositePort, +ListByPort,
  *       -ResultingList)
  *
  * Create a list of ui definition associated to each given composite ports
  *
  * Arg 3 is the resulting list which looks like :
  * [[Port,[[UIDefinition],...]],...]
  *
  */
get_all_associated_port_with_ui_definition([],_List,[]).
get_all_associated_port_with_ui_definition([[PortCC,PortOfSub]|Others],List,
                                           [[PortCC,ResultingList]|OtherCorresponding]):-
        all_associated_port_with_ui_definition(PortCC,PortOfSub,List,ResultingList),
        get_all_associated_port_with_ui_definition(Others,List,OtherCorresponding),!.
```

Listing B.29 – Prédicat permettant d'obtenir une liste qui associe à chacun des ports d'entrée du composant composite l'ensemble des représentation graphique

Le listing B.30 présente le prédicat *all_associated_port_with_ui_definition* qui est définit avec quatre et cinq arguments. Celui qui comporte quatre arguments est appelé par le précédent prédicat et permet d'itérer sur chacun des couples *[sous-composant,ports liés]*. Le prédicat qui possède cinq paramètres permet d'itérer sur la liste des ports liés associé à un sous-composant et à un port du composant composite. Celui-ci fait appel au prédicat *associated_port_with_ui_definition* qui va permettre d'obtenir la description associée.

```
/**
  *
  * PRED: all_associated_port_with_ui_definition( +Port, +ListAssociatedComponentPort,
  *       +ListByPort, ?ResultingList)
  *
  * Create a list of ui definition
  *
  * Arg 4 is the resulting list which looks like :
  * [[UIDefinition],[UIDefinition],...]
  *
  */
all_associated_port_with_ui_definition(_PortCC, [],_List,[]).
all_associated_port_with_ui_definition(PortCC,[[SubComponent,Ports]|Other],List,
                                       ResultingList):-
        all_associated_port_with_ui_definition(PortCC,SubComponent,Ports,List,ListDesc),
        all_associated_port_with_ui_definition(PortCC, Other,List,OtherListDesc),!,
        append(ListDesc,OtherListDesc,ResultingList).
all_associated_port_with_ui_definition(_PortCC,_SubComponent,[],_List,[]).
all_associated_port_with_ui_definition(PortCC,SubComponent,[PortSC|Other],List,
                                       [UIDesc|OtherDesc]):-
        associated_port_with_ui_definition(PortCC,SubComponent,PortSC,List,UIDesc),
        all_associated_port_with_ui_definition(PortCC,SubComponent,Other,List,OtherDesc),!.
```

Listing B.30 – Prédicat permettant d'obtenir une liste de l'ensemble des représentation graphique

Le listing B.31 présente le prédicat *associated_port_with_ui_definition*. Celui-ci permet d'obtenir la description graphique associée à un port d'un sous-composant. Pour trouver la bonne description, on itère sur la liste qui associe la représentation graphique au port d'un composant fonctionnel. Dès que l'élément est trouvé, on arrête l'itération. Cette itération s'arrête dès que le prédicat *get_definition* trouve l'élément correspondant.

```
/**
  *
  * PRED: associated_port_with_ui_definition( +Port, +SubComponent, +Port, +ListComposantUIInfo,
  *       -UIDescription)
  *
  * Obtain the good ui definition
  *
  * Arg 5 is the resulting list which looks like :
  * [UIDefinition]
  *
  */
associated_port_with_ui_definition(PortCC, SubComponent, PortSC,
                                   [[SubComponent,Info]|_Others], UIDesc):-
        get_definition(PortCC,SubComponent,PortSC,Info,UIDesc),!.
associated_port_with_ui_definition(PortCC, SubComponent, PortSC,
                                   [[_OtherComponent,_Info]|Others],UIDesc):-
        associated_port_with_ui_definition(PortCC, SubComponent, PortSC, Others,UIDesc).
```

Listing B.31 – Prédicat permettant d'obtenir une représentation graphique

Le listing B.32 présente le prédicat *get_definition* qui permet d'obtenir la description d'IHM associée à un port. Ce prédicat est utilisé dans le prédicat précédent afin de déterminer si l'élément est présent dans le couple : *[sous-composant,information]* (*information* correspondant à l'ensemble des couples *[port, description graphique]*).

```
/**
  *
  * PRED: get_definition( +Port, +SubComponent, +Port, +ListUIInfo, ?UIDescription)
  *
  * Obtain the good ui definition
  *
```

```
  * Arg 5 is the resulting list which looks like :
  * [UIDefinition]
  *
  */
get_definition(_PortCC,_SubComponent,PortSC,[[PortSC,UIDesc]|_Others],UIDesc).

get_definition(PortCC,SubComponent,PortSC,[[_OtherPortSC,_UIDesc]|Others],UIDesc):-
        get_definition(PortCC,SubComponent,PortSC,Others,UIDesc).
```

Listing B.32 – Prédicat permettant d'obtenir la représentation graphique

B.3 Seconde étape : détermination des points de conflits

Pour permettre de déterminer les points de conflits, il est nécessaire de réaliser des étapes intermédiaires. Ces étapes sont :

1. Déterminer les points de fusion
2. Extraire les points de fusion de la liste qui associe à chaque port l'ensemble des représentations graphiques
3. Déterminer pour chacun des points de fusion si c'est un point de conflit et d'où provient le conflit
4. Supprimer tous les conflits qui sont inconsistants (dans le sens où une représentation graphique qui a une cardinalité *n* ne peut être liée à un élément du fonctionnel qui prend qu'une valeur).

B.3.1 Déterminer les points de fusion

Pour ce faire, on a trois prédicats qui permettent de déterminer les points de fusion au niveau des entrées, des sorties et des actions (pour les actions cela signifie que plusieurs opérations sont appelées en même temps). Ce qui suit présente le prédicat associé aux entrées, mais la démarche reste similaire pour la recherche des points de fusion pour les sorties et les actions.

Le listing B.33 permet de calculer la liste des points de fusion en entrée. Pour ce faire, on passe par une étape intermédiaire qui permet d'obtenir une liste qui contient des listes de ports d'entrée du composant composite. Cette liste est calculée en itérant sur chacun des sous-composants présents au sein du composant composite et en regardant les liens de données associés. Cette liste est ensuite utilisée dans le prédicat *merged_port* qui va calculer la liste des entrées qui sont fusionnées dans le cas présent.

```
/**
  *
  * PRED: merged_int_res( +CompositeFC, +SubComponents, ?List)
  *
  * Create a list of merged in port (which are connected or more than
  * one port)
  *
  * Arg 3 is the result of the research of merged ports
  *
  */
merged_in_res(ComposeFC,SubComponents,MergedIn):-
        merged_in(ComposeFC,SubComponents,TempList),
        merged_port(TempList,MergedIn).
merged_in(_ComposeFC,[],[]).
merged_in(ComposeFC,[SubComponent|ListSubComponents],[PortIn|OtherPort]):-
        data_link(composite_fc(ComposeFC,_,_,_,_),SubComponent,ListLink),
        get_first_element(ListLink,PortIn),
        merged_in(ComposeFC,ListSubComponents,OtherPort),!.
merged_in(ComposeFC,SubComponent,[PortIn]):-
        data_link(composite_fc(ComposeFC,_,_,_,_),SubComponent,ListLink),
        get_first_element(ListLink,PortIn).
```

Listing B.33 – Prédicats qui permettent d'obtenir la liste des points de fusion

Le listing B.3.1 présente le prédicat *merged_port* mais également le prédicat *temp_merged_port* qui va réaliser l'intersection des éléments présents dans la liste précédemment calculée. C'est une intersection deux à deux qui est réalisée afin de trouver les éléments qui sont présents dans au moins deux listes de la liste fournie en entrée. L'utilisation du prédicat *sort* permet de trier et de supprimer les doublons de la liste obtenue. Cette liste contient ainsi l'ensemble des points de fusion.

```
/**
  *
  * PRED: temp_merged_port( +ListSource, ?List)
  *
  * Create a list of intersection elements that correspond to merged element
  *
  * Arg 2 is the resulting list
  *
  */
temp_merged_port([],[]).
temp_merged_port([A,B|Queue],[Res1,Res2,Intersection]):-
        temp_merged_port([B|Queue],Res1),
        temp_merged_port([A|Queue],Res2),
        intersection(A,B,Intersection),!.
temp_merged_port([A,B],Intersection):-
        intersection(A,B,Intersection).

merged_port(List,Res):-
        temp_merged_port(List,TempRes),!,
        length(TempRes,Size),
        (Size > 1 -> append(TempRes,TempRes1),sort(TempRes1,Res) ;
            sort(TempRes,Res)
        ).
```

B.3.2 Extraction des points de fusion

Avant de contrôler si les points de fusion sont des points de conflits ou pas, on extrait de chacune des listes : *AssociatedFCPortInWithUI*, *AssociatedFCPortOutWithUI* et *AssociatedFCPortActionWithUI* deux sous-listes qui séparent d'un côté les éléments non fusionnés et de l'autre les éléments fusionnés. Pour ce faire, on utilise le prédicat *extract_all_merged_element* qui permet la création des deux sous-listes à partir d'une liste des éléments fusionnés et d'une liste qui associe aux ports du composant composite l'ensemble des représentations graphiques.

Le listing B.34 présente ce prédicat. On voit que lorsqu'il n'y a aucun point de fusion, le résultat est direct puisque la liste sans point de fusion correspond à la liste en entrée alors que la liste avec les points de fusion, correspond à une liste vide. Pour chacun des points de fusion, on fait appel au prédicat *find_good_element* qui permet de trouver l'élément dans la liste d'entrée. Cet élément est supprimé de la liste afin de pouvoir obtenir la liste définitive des éléments non fusionnés et est ajouté à la liste contenant les éléments fusionnés.

```
/**
  *
  * PRED: extract_all_merged_element( +ListPort, +ListPortDes, -ResultingList, -MergedList)
  *
  * Create two new lists that correspond to :
  * 1) one without merged element Arg 3
  * 2) one with merged element Arg 4
  *
  */
extract_all_merged_element([],ListOriginal,ListOriginal,[]).
extract_all_merged_element([Port|Other],ListOriginal,ResultingList,[Elem|Res]):-
        extract_all_merged_element(Other,ListOriginal,TmpList,Res),
        find_good_element(Port,TmpList,Elem),
        delete(TmpList,Elem,ResultingList),!.
```

Listing B.34 – Prédicat qui permet de séparer une liste associant port à représentation graphique en deux sous-listes : une avec les points de fusion et une sans

Le listing B.35 présente le prédicat *find_good_elemet*. Ce prédicat permet de trouver l'élément recherché au sein d'une liste de couple. Le premier élément correspond à une clé et le second à la valeur recherchée. Dans ce cas, le but est d'extraire l'ensembles des descriptions graphiques associé à un port. C'est cet élément qui est utilisé dans le prédicat précédent pour le supprimer de la liste d'origine et l'ajouter à la liste des éléments fusionnés.

```
/**
  *
  * PRED: find_good_element( +Port, +ListPortDes, -Desc)
  *
  * Create a list of ui definition
  *
  * Arg 3 is the description that corresponds to the port
  * [[UIDefinition],[UIDefinition],...]
  *
  */
find_good_element(_Port,[],[]).
find_good_element(Port,[[Port,List]|_OtherPorts],[Port,List]).
find_good_element(Port,[[_OtherPort,_List]|OtherPorts],List):-
        find_good_element(Port,OtherPorts,List),!.
```

Listing B.35 – Prédicat qui permet de trouver le bon élément dans une liste de couple

Le résultat ainsi obtenu à la suite de cette exécution est six listes qui pour chacun des types de ports (entrée, sortie et action) associe deux listes qui contiennent dans l'une les éléments fusionnés et dans l'autre les éléments non fusionnés.

La liste qui contient les éléments fusionnés est utilisée dans la suite pour déterminer si c'est un point de conflit ou non.

B.3.3 Déterminer les points de conflits

Pour déterminer si un point fusionné est un point de conflit, il est nécessaire de comparer l'ensemble des éléments qui compose la représentation graphique. Ainsi dans le cas d'une entrée, il faut comparer si le nom, le type, la cardinalité et la sélection sont identiques ; dans le cas d'une sortie, si le nom, le type et la cardinalité sont identiques ; et dans le cas d'une action si le nom est identique.

La comparaison est faite deux à deux sur l'ensemble des éléments fusionnés et utilise pour ce faire le prédicat *test* qui compare deux éléments. On obtient les conflits qui existent (en donnant les valeurs conflictuelles) ou dans le cas où il n'y a pas de conflit une liste vide pour chacune des valeurs comparées. Le listing B.36 présente le prédicat *test* ainsi que l'ensemble des prédicats de comparaison.

```
/**
  *
  * PRED: test( +Elem1, +Elem2, -Result)
  *
  * Compare Elem1 and Elem2 attributes
  * Result of the comparison in Result (Arg 3)
  * If no conflict: empty list
  *
  */
test([_UIComponent1,_UIPort1,UIName1],[_UIComponent2,_UIPort2,UIName2],ResName):-
        test_name([UIName1,UIName2],ResName).

test([_UIComponent1,_UIPort1,UIName1,UIType1,UIArity1],
     [_UIComponent2,_UIPort2,UIName2,UIType2,UIArity2],
     ResName,ResType,ResArity):-
        test_name([UIName1,UIName2],ResName),
        test_type([UIType1,UIType2],ResType),
        test_arity([UIArity1,UIArity2],ResArity).

test([_UIComponent1,_UIPort1,UIName1,UIType1,UIArity1,UISelection1],
     [_UIComponent2,_UIPort2,UIName2,UIType2,UIArity2,UISelection2],
     ResName,ResType,ResArity,ResSelection):-
        test_name([UIName1,UIName2],ResName),
```

```
        test_type([UIType1,UIType2],ResType),
        test_arity([UIArity1,UIArity2],ResArity),
        test_arity([UISelection1,UISelection2],ResSelection).

test_name([Name1,Name1],[]).
test_name([Name1,Name2],[Name1,Name2]).

test_type([Type1,Type1],[]).
test_type([Type1,Type2],[Type1,Type2]).

test_arity([Arity1,Arity1],[]).
test_arity([Arity1,Arity2],[Arity1,Arity2]).

test_selection([Selection1,Selection1],[]).
test_selection([Selection1,Selection2],[Selection1,Selection2]).
```

Listing B.36 – Prédicats qui permettent de tester si deux éléments sont identiques

Le listing B.37 présente le prédicat qui permet de comparer l'ensemble des représentations graphiques associé à un port. Cette comparaison est réalisée deux à deux sur chacun des éléments présents. Le résultat obtenu est une liste qui pour chacune des caractéristiques de la représentation graphique contient les valeurs en conflit. Dans le cas où il n'y a pas de conflits détectés, la liste obtenue est vide. Le prédicat présenté ici permet de tester les représentations graphiques associées à un port d'entrée. Par ailleurs, ce prédicat peut posséder un nombre différent d'arguments afin de pouvoir réaliser la comparaison des représentations graphiques associées à une sortie ou à une action.

```
/**
  *
  * PRED: comparison_of_merged_port( +ListOfMergedInput, -ListOfNameConflict,
  *       -ListOfTypeConflict, -ListOfArityConflict, -ListOfSelectionConflict)
  *
  * Create a list that contain name conflict element (Arg 2)
  * Create a list that contain type conflict element (Arg 3)
  * Create a list that contain arity conflict element (Arg 4)
  * Create a list that contain selection conflict element (Arg 5)
  *
  */
comparison_of_merged_port([],[],[],[],[]).
comparison_of_merged_port([Elem1,Elem2|OtherElem],ResultingName,ResultingType,ResultingArity,
                          ResultingSelection):-
        test(Elem1,Elem2,ResName,ResType,ResArity,ResSelection),
        comparison_of_merged_port([Elem1|OtherElem],ResName1,ResType1,ResArity1,ResSelection1),
        comparison_of_merged_port([Elem2|OtherElem],ResName2,ResType2,ResArity2,ResSelection2),
        append(ResName,ResName1,TempResName),
        append(TempResName,ResName2,ResultingName),
        append(ResType,ResType1,TempResType),
        append(TempResType,ResType2,ResultingType),
        append(ResArity,ResArity1,TempResArity),
        append(TempResArity,ResArity2,ResultingArity),
        append(ResSelection,ResSelection1,TempResSelection),
        append(TempResSelection,ResSelection2,ResultingSelection),!.
comparison_of_merged_port([Elem1,Elem2],ResName,ResType,ResArity,ResSelection):-
        test(Elem1,Elem2,ResName,ResType,ResArity,ResSelection).
```

Listing B.37 – Prédicat qui réalise une comparaison de chacun des éléments deux à deux

Le listing B.38 présente le prédicat *comparison_of_all_merged_port_input* qui réalise une itération sur l'ensemble des éléments (en entrée) présents dans la liste des éléments fusionnés. Pour chacun des éléments présents, on fait appel au prédicat *comparison_of_merged_port* précédemment présenté. Une fois le résultat obtenu, on applique un *sort* sur chacune des listes obtenues afin de supprimer les doublons (vu que la comparaison se fait deux à deux). Si aucun conflit n'a été détecté, l'élément n'est pas ajouté à la liste des conflits, dans le cas contraire, une liste est créée qui contient le nom du port du composant composite avec l'ensemble des conflits associés.

```
/**
  *
  * PRED: comparison_of_all_merged_port_input( +ListOfMergedInput, -ListOfConflict)
  *
```

```
  * Create a list that contain conflict element (Arg 2)
  * Empty list if no conflict
  *
  */
comparison_of_all_merged_port_input([],[]).
comparison_of_all_merged_port_input([[Port,MergedElements]|Others],FinalResult):-
        comparison_of_merged_port(MergedElements,TempResName,TempResType,TempResArity,
                                  TempResSelection),
        sort(TempResName,ResName),
        sort(TempResType,ResType),
        sort(TempResArity,ResArity),
        sort(TempResSelection,ResSelection),
        length(ResName,Size1),
        length(ResType,Size2),
        length(ResArity,Size3),
        length(ResSelection,Size4),
        ((Size1 < 1,Size2 < 1, Size3 <1, Size4 < 1) -> append([],OtherPb,FinalResult) ;
            append([[Port,ResName,ResType,ResArity,ResSelection]],OtherPb,FinalResult)),
        comparison_of_all_merged_port_input(Others,OtherPb),!.
```

Listing B.38 – Prédicat qui permet de réaliser la comparaison de chacun des éléments

Le prédicat *conflict_detection_input* présenté dans le listing B.39 a pour objectif de détecter l'ensemble des points de conflits en s'appuyant sur l'ensemble des prédicats présentés auparavant. Deux entrées sont utilisées pour ce prédicat qui sont la liste des points de fusion et la liste sans point de fusion. La première liste est utilisée dans la détection. La seconde est utilisée en fin de traitement afin de remettre l'ensemble des points de fusion qui ne sont pas considérés comme des points de conflits et qui sont donc identiques. Le résultat obtenu à la suite de l'appel au prédicat *comparison_of_all_merged_port_input* est une liste qui contient l'ensemble des points de conflits sous la forme (pour les entrées) : $[[PortIn, [ConflitName], [ConflitType], [ConflitArity], [ConflitSelection]], ...]$. Le premier post-traitement qui est fait permet de recréer les listes correctes qui correspondent cette fois-ci aux éléments en conflits et aux éléments sans conflits avec pour chacune d'elle associé à un port la représentation ou les représentations graphiques. Le second post-traitement permet de ne conserver qu'une seule représentation pour les éléments qui sont sans conflits.

```
/**
  *
  * PRED: conflict_detection_input( +ListOfMerged, +ListWithoutMerged, -ListWithConflict,
  *       -ListWithoutConflict, -ListOfConflict)
  *
  * Create a three lists that contain the final list of conflict elements
  * (Arg 3),the final list of non-conflict element (Arg 4)
  * and the list of conflict (Arg 5)
  *
  */
conflict_detection_input(ListOfMerged,ListWithoutMerged,ListWithConflict,ListWithoutConflict,
                         ListOfConflict):-
        comparison_of_all_merged_port_input(ListOfMerged,ListOfConflict),
        post_treatment(ListOfConflict,ListOfMerged,TmpListWithoutConflict,ListWithConflict),
        post_treatment_only_one(TmpListWithoutConflict,ListWithOneUIElem),
        append(ListWithoutMerged,ListWithOneUIElem,ListWithoutConflict).

/**
  *
  * PRED: post_treatment( +ListOfConflicts, +ListOfMerged, -ListWithoutConflict,
  *       -ListWithConflict)
  *
  * Create a two lists that contain the final list of conflict elements
  * (Arg 4) and a temporary list of non-conflict element (Arg 3)
  *
  */
post_treatment([],ListOfMerged,ListOfMerged,[]).
post_treatment([[Port|_]|Other],ListOfMerged,NewListWithoutConflict,NewListOfConflict):-
        post_treatment(Other,ListOfMerged,TmpList,TmpList1),
        find_good_element(Port,TmpList,Elem),
        delete(TmpList,Elem,NewListWithoutConflict),
        append(TmpList1,[Elem],NewListOfConflict).

/**
```

```
  *
  * PRED: post_treatment_only_one( +ListOfMergedWithoutConflict, -NewListWithOnlyOne)
37  *
  * Create a list that contain only one element for the merged elements (Arg 2)
  *
  */
post_treatment_only_one([],[]).
42 post_treatment_only_one([[Port,[Elem|_OtherElem]]|OtherPort],[[Port,[Elem]]|OtherPortPost]):-
        post_treatment_only_one(OtherPort,OtherPortPost),!.
```

Listing B.39 – Prédicat qui permet de détecter les points de conflits et qui fait appel au prédicat de comparaison

A partir de la liste obtenue qui contient l'ensemble des points de conflit, on ajoute une nouvelle étape qui permet de vérifier la consistence des éléments d'IHM associés au port du composant composite. Cette étape n'est appliquée que sur les ports de données.

B.3.4 Suppression de conflits par inconsistence

L'objectif ici est de pouvoir supprimer un certain nombre de conflit lorsque les éléments graphiques associés ne sont pas consistent par rapport au port fonctionnel auquel ils sont associés. Cela se produit lorsqu'une entrée du composant composite possède une cardinalité $[0,1]$ ou $[1,1]$ et est associé à un élément graphique de cardinalité n. Il n'est alors pas possible de fournir une liste en entrée à part si celle-ci possèdent une sélection à $SINGLE$ (qui fait qu'un seul élément de la liste sera utilisé). La suppression des éléments inconsistents se base principalement sur le prédicat *find_all_by_arity* qui permet de trouver les éléments graphiques qui sont liés à un port de données en conflit et qui extrait que les descriptions graphiques qui permettent l'usage de ce port. Pour ce faire, une table de recherche est mise en place (cf. table B.1). Cette table permet de savoir quels sont les éléments qui doivent être conservés pour être utilisés avec le port d'entrée ou de sortie auxquels ils sont liés.

	Cardinalité pour le port fonctionnel	Cardinalité pour la représentation graphique
Port d'entrée	1 n	1 ou n (si sélection=*single*) 1 ou n
Port de sortie	1 n	1 ou n n

TABLE B.1 – Table de consistence pour les associations entre port fonctionnel et représentation graphique

Le listing B.40 permet d'obtenir l'ensemble des représentations graphiques qui sont consistants avec la cardinalité du port fonctionnel auxquels elles sont associées. De cette liste ainsi obtenu il est possible de savoir si l'élément est toujours en conflit ou pas. En effet, si un seul élément est présent dans la liste obtenue, alors il n'y a plus de conflits et donc l'élément doit être placé dans la liste des éléments sans conflit. Si la liste est vide, cela signifie qu'aucun élément n'est consistent par rapport au port fonctionnel auquel il est attaché. Le conflit persiste mais il advient à l'utilisateur de le résoudre en ajoutant lui-même l'information nécessaire. Enfin, s'il existe plus d'un élément qui est consistent alors le conflit persiste. Une fois cette liste obtenue, un post-traitement doit être effectué afin d'obtenir : (i) la liste qui contient les conflits, (ii) la liste qui contient l'association entre ports avec conflit du composant composite et représentation graphique et (iii) la liste qui contient l'association entre ports sans conflit du composant composite et représentation graphique. Le but de ce post-traitement est de remettre à jour les listes si nécessaire est avant de passer à l'étape suivante du processus de composition.

```
/**
  *
  * PRED: find_all_by_arity( +FCArity, +ListOfMerged, -ConstraintList)
  *
  * The goal is to obtain the list that contain elements that
  * match with the arity (Arg 3)
  *
  */
find_all_by_arity(_FCArity,[],[]).
find_all_by_arity([_,1],
                  [[ComponentAUI,PortID,Name,Type,Arity]|Others],
                  [[ComponentAUI,PortID,Name,Type,Arity]|OthersGood]):-
        find_all_by_arity([_,1],Others,OthersGood),!.
find_all_by_arity([_,1],
                  [[ComponentAUI,PortID,Name,Type,1,Selection]|Others],
                  [[ComponentAUI,PortID,Name,Type,1,Selection]|OthersGood]):-
        find_all_by_arity([_,1],Others,OthersGood),!.
find_all_by_arity([_,1],
                  [[ComponentAUI,PortID,Name,Type,n,single]|Others],
                  [[ComponentAUI,PortID,Name,Type,n,single]|OthersGood]):-
        find_all_by_arity([_,1],Others,OthersGood),!.
find_all_by_arity([_,1],
                  [[_ComponentAUI,_PortID,_Name,_Type,_,_Selection]|Others],
                  OthersGood):-
        find_all_by_arity([_,1],Others,OthersGood),!.
find_all_by_arity([_,n],
                  [[ComponentAUI,PortID,Name,Type,n]|Others],

                  [[ComponentAUI,PortID,Name,Type,n]|OthersGood]):-
        find_all_by_arity([_,n],Others,OthersGood),!.
find_all_by_arity([_,n],
                  [[_ComponentAUI,_PortID,_Name,_Type,_]|Others],
                  OthersGood):-
        find_all_by_arity([_,n],Others,OthersGood),!.
find_all_by_arity([_,n],
                  [[ComponentAUI,PortID,Name,Type,Arity,Selection]|Others],
                  [[ComponentAUI,PortID,Name,Type,Arity,Selection]|OthersGood]):-
        find_all_by_arity([_,n],Others,OthersGood),!.
```

Listing B.40 – Prédicat qui permet de rechercher les éléments qui sont cohérent avec le port fonctionnel

A la fin de cette étape, si aucun élément est en conflit, il est possible de passer directement à la quatrième étape du processus de composition. Dans le cas où il y a des conflits, il est nécessaire que le développeur les résolve avant la création des composants d'IHM. Pour ce faire, il est nécessaire de réaliser la troisième étape.

B.4 Troisième étape : résolution des points de conflits

La troisième étape consiste en la résolution des conflits. Cette résolution des conflits est réalisée par l'intervention du développeur qui est en train de procéder à la composition. Plusieurs possibilités sont offertes à lui. En effet, il peut :

- choisir une représentation graphique parmi l'ensemble des représentations présentes ;
- choisir de conserver l'ensemble des représentations graphiques présentes ;
- sélectionner des informations en provenance de différentes représentations graphiques ;
- personnaliser complètement le résultat en fournissant lui-même les informations pour obtenir une nouvelle représentation graphique.

L'objectif à la fin de l'étape de résolution de conflits est de compléter la liste des éléments sans conflits afin de finaliser le processus de composition et d'ainsi pouvoir créer l'ensemble des composants d'IHM associés à la composition fonctionnelle.

Pour aider le développeur dans la résolution des conflits, un ensemble d'informations lui est fourni. Ces informations présentent au développeur le port du composant fonctionnel sur lequel il y a un conflit, sur quelles informations se trouve le conflit et enfin les différentes possibilités of-

fertes pour résoudre le conflit. Ces informations sont nécessaires pour une meilleure réutilisation des informations déjà présentes.

Il existe quatre conflits différents à résoudre qui sont :

- les conflits de dénomination. Les éléments graphiques ne possèdent pas le même nom et donc il est nécessaire de déterminer quel est le nom qui doit être conservé.
- les conflits de types. Les éléments graphiques possèdent des types différents. Il est alors intéressant de prendre celui qui peut convenir le mieux pour l'usage du composant composite.
- les conflits de cardinalité. Si les éléments n'ont pas la même cardinalité, il faut décider si l'on souhaite conserver une liste ou un élément unaire.
- les conflits de sélection. Lorsque deux éléments représentent des listes mais qu'ils ne possèdent pas la même liberté de sélection, laquelle doit alors être conservée.

Afin de laisser une plus grande liberté à l'utilisateur sur la résolution des conflits, le choix de la politique de résolution de conflit est effectué sur chacun des ports qui possèdent un conflit.

La résolution des conflits se fait au travers de l'usage d'une extra-IHM [Sot08]. Cette IHM permet au développeur d'avoir une vision globale des conflits et de fournir une interface de contrôle qui permet de sélectionner pour chacun des conflits comment celui-ci doit être résolu.

B.5 Quatrième étape : création des composants d'IHM et de l'application

La création des composants d'IHM est réalisée en se basant sur l'ensemble des informations précédemment obtenues. Les listes utilisées associent à chaque port sa ou ses représentations graphiques. En effet, il est possible de conserver plusieurs représentations graphiques pour un port, à ce moment là, un lien de cohérence est mis en place entre les différents éléments graphiques pour être sûr que l'information est identique dans chacun des éléments. Toutes les listes utilisées ne contiennent plus de conflits, ceux-ci ont été résolus dans l'étape précédente.

L'objectif est de créer pour chacune des actions présentes un composant d'IHM. Pour ce faire on utilise le prédicat *realize_ui_components* qui va faire appel au prédicat *create_ui_component* qui est en charge d'itérer sur l'ensemble des actions présentes au sein du composant composite et qui va créer les composants d'IHM associés.

Le listing B.41 présente le premier prédicat qui est le déclencher de la création de l'ensemble des composants d'IHM. Le résultat obtenu suite à l'appel à ce prédicat est une liste qui contient l'ensemble des composants d'IHM créés ainsi que l'ensemble des définitions de ports, des liens de données entre entrées et entre sorties, des liens d'événement et pour finir des liens internes pour permettre de conserver la consistance. Ces listes sont ensuite utilisées pour finaliser la concrétisation des composants d'IHM créés pour obtenir une application utilisable.

Pour chacune des listes créées, voici la forme qu'elles ont :

- pour la liste des composants d'IHM : $[[nom\ composant\ d'IHM, [IDPortsIn], [IDPortsOut], [IDPortsEvent]], ...]$,
- pour la liste de définition des ports : $[[nom\ composant\ d'IHM, [[idPortIn, nom, type, arite, selection], ...], [[idPortOut, nom, type, arite], ...], [[idPortEvent, nom, [portsIn], [portsOut]], ...]], ...]$
- pour les liste de liens : $[[composant\_source, composant\_destination, [[port\_source, port\_destination], ...]], ...]$. Dans le cas de la liste des liens internes le composant source et destination sont identiques et correspondent à un composant d'IHM qui possède un tel lien de cohérence.

```
realize_ui_components(CompositeFC,[ListOfUIin,ListOfUIout,ListOfUIaction],
                      [ListOfUIComponent,ListOfPortDefinition,ListOfDataLinkII,
                       ListOfDataLinkOO,ListOfEventLink,ListOfInternalLink]):-
    composite_fc(CompositeFC,_,_,_,PortAction),
```

```
        create_ui_component(CompositeFC,PortAction,ListOfUIin,ListOfUIout,
                            ListOfUIaction,ListOfUIComponent,
                            ListOfPortDefinition,ListOfDataLinkII,
                            ListOfDataLinkOO,ListOfEventLink,ListOfInternalLink).
```

Listing B.41 – Prédicat qui lance la création de l'ensemble des composants d'IHM

Le listing B.42 présente le prédicat qui est le chef d'orchestre de la réalisation de la création des composants d'IHM. En effet, c'est lui qui appelle les différents prédicats nécessaires à la création des ports de données et d'événement. Les liens sont réalisés en même temps que les ports sont créés. A partir de là, il est alors possible de créer l'ensemble des éléments qui vont dans les différentes listes précédemment énoncées. Le nom donné au composant d'IHM est obtenu de manière unique en utilisant le nom du composant composite et en lui ajoutant "_aui" et un numéro. On itère ici sur la liste des actions présentes dans le composant composite afin de créer autant de composants d'IHM qu'il n'y a de ports d'action. Pour créer les ports (à partir des informations dont on dispose), on fait appel à trois prédicats pour chacun des types de ports à créer. Lors de la création de ces ports, on crée également une liste de couple qui met en relation un port du composant d'IHM qui est en train d'être créé avec le port du composant composite auquel il est associé.

```
create_ui_component(_Composite,[],_ListOfUIin,_ListOfUIout,_ListOfUIaction,[],[],[],[],[],[]):-
        reset_gensym('ui_input'),reset_gensym('ui_output'),reset_gensym('ui_action').
create_ui_component(Composite,[PortAction|OtherPortAction],
                    ListOfUIin,ListOfUIout,ListOfUIaction,
                    [UIComponent|ListOfUIComponent],
                    [AllPortsDefinition|ListOfPortDefinition],
                    [DataLinkII|ListOfDataLinkII],
                    [DataLinkOO|ListOfDataLinkOO],
                    [EventLink|ListOfEventLink],
                    [InternalLink|ListOfInternalLink]):-
        event_data_association(composite_fc(Composite,_,_,_,_),PortAction,_Name,ListIn,ListOut),
        create_ui_component_in_port(ListIn,ListOfUIin,ListPortIdIn,
                                    ResultingPortIn,ResultingDataLinkIn,
                                    ResultingDataLinkInternalIn),
        create_ui_component_out_port(ListOut,ListOfUIout,ListPortIdOut,
                                     ResultingPortOut,ResultingDataLinkOut,
                                     ResultingDataLinkInternalOut),
        create_ui_component_action_port(PortAction,ListOfUIaction,ListPortIdIn,
                                        ListPortIdOut,ResultingPortAction,
                                        ResultingEventLink,ListPortIdAction),
        create_ui_component_name(Composite,UIComponentName),
        UIComponent=[UIComponentName,ListPortIdIn,ListPortIdOut,ListPortIdAction],
        AllPortsDefinition=[UIComponentName,ResultingPortIn,
                            ResultingPortOut,ResultingPortAction],
        DataLinkII=[UIComponentName,Composite,ResultingDataLinkIn],
        DataLinkOO=[Composite,UIComponentName,ResultingDataLinkOut],
        EventLink=[UIComponentName,Composite,ResultingEventLink],
        append(ResultingDataLinkInternalIn,ResultingDataLinkInternalOut,
               ResultingDataLinkInternal),
        InternalLink=[UIComponentName,UIComponentName,ResultingDataLinkInternal],
        reset_gensym('ui_input'),
        reset_gensym('ui_output'),
        reset_gensym('ui_action'),
        create_ui_component(Composite,OtherPortAction,ListOfUIin,ListOfUIout,
                            ListOfUIaction,ListOfUIComponent,
                            ListOfPortDefinition,ListOfDataLinkII,
                            ListOfDataLinkOO,ListOfEventLink,
                            ListOfInternalLink),!.

create_ui_component_name(Composite,UIComponent):-
        concat_atom([Composite,'aui'],'_',TempId),
        gensym(TempId,UIComponent).
```

Listing B.42 – Prédicat qui réalise la création des composants d'IHM

B.5.1 Création des ports de données

Ce qui suit présente la création des ports d'entrée. Le mécanisme utilisé est le même pour la création des ports de sorties. Pour ce faire, on itère sur l'ensemble des ports d'entrée du composant composite qui est associé à l'action qui est en train d'être traitée. On utilise également la liste qui associe à chaque port du composant composite la représentation graphique. Pour commencer, on cherche à obtenir l'ensemble des informations graphiques associées au port qui est traité. Ces informations sont utilisées par le prédicat *create_ui_port_in* (cf. listing B.43) qui se charge de créer l'ensemble des ports d'entrée. Deux listes résultent qui contiennent pour l'une, l'ensemble l'ensemble des identifiants des ports créés et pour l'autre, l'ensemble des définitions. La liste qui contient l'ensemble des identifiants est utilisée pour créer les liens internes (si nécessaires) et les liens de données.

```
create_ui_component_in_port([],_ListOfUIin,[],[],[],[]).
create_ui_component_in_port([PortIn|OtherPort],ListOfUIin,
                            ResultingListPortIdIn,ResultingPortIn,
                            ResultingDataLink,ResultingDataLinkInternal):-
        find_good_element(PortIn,ListOfUIin,[_,ElemIn]),
        create_ui_port_in(PortIn,ElemIn,ListPortIdIn,PortInfo),
        create_internal_link(ListPortIdIn,ListInternalLinkIn),
        create_data_link_in(PortIn,ListPortIdIn,ListLink),
        create_ui_component_in_port(OtherPort,ListOfUIin,
                                    TmpListPortIdIn,TmpResultingPortIn,
                                    TmpResultingDataLink,
                                    TmpResultingDataLinkInternal),!,
        append(ListPortIdIn,TmpListPortIdIn,ResultingListPortIdIn),
        append(PortInfo,TmpResultingPortIn,ResultingPortIn),
        append(ListLink,TmpResultingDataLink,ResultingDataLink),
        append(ListInternalLinkIn,TmpResultingDataLinkInternal,ResultingDataLinkInternal).
```

Listing B.43 – Création des ports d'entrée

La création des ports de données se fait par itération sur l'ensemble des descriptions associées à un port du composant composite. C'est le prédicat *create_ui_port_in* (cf. listing B.44) qui se charge de cette création. Il prend en entrée deux arguments qui sont le port d'entrée du composant fonctionnel qui est traité ainsi que la liste des descriptions graphiques associées à celui-ci. Pour chacune des descriptions, on crée un port d'entrée auquel on associe un identifiant unique au sein du composant créé. Le résultat obtenu est deux listes qui ont pour forme :

- $[IdPortIn, IdPortIn, ...]$
- $[[IdPortIn, Name, Type, Arity, Selection], ...]$

```
create_ui_port_in(_PortIn,[],[],[]).
create_ui_port_in(PortIn,[[_CUI,_ID,Name,Type,Arity,Selection]|OtherIn],
                  [PortID|OtherID],ResListPortInfo):-
        gensym('ui_input',PortID),
        TmpPortInfo=[PortID,Name,Type,Arity,Selection],
        create_ui_port_in(PortIn,OtherIn,OtherID,ListPortInfo),
        append([TmpPortInfo],ListPortInfo,ResListPortInfo).
```

Listing B.44 – Création de la définition des ports d'entrée

La création des liens internes (cf. listing B.45) n'est réalisée que s'il y a plus d'un port d'entrée créé pour un port du composant fonctionnel. Dans ce cas, on crée un ensemble de liens internes qui permettent de conserver la cohérence dans les valeurs qu'ils possèdent. Ainsi, lorsque l'utilisateur saisit une valeur, celle-ci doit être répercutée sur l'ensemble des ports (donc des composants graphiques) qui sont liés entre eux. Les liens créés ne sont pas orientés comme c'est le cas des autres liens. Pour la réalisation de ces liens on itère deux à deux sur les ports précédemment créés.

```
create_internal_link([],[]).
create_internal_link([_],[]).
create_internal_link([A,B|Other],Link):-
        create_internal_link([A|Other],Temp1),
        create_internal_link([B|Other],Temp2),
```

```
        append([[A,B]],Temp1,Temp),
        append(Temp,Temp2,Link),!.
8 create_internal_link([A,B],[[A,B]]).
```

Listing B.45 – Création des liens internes

Pour finir, la réalisation des liens de données est effectuée. Dans le cas où il y a plusieurs ports d'entrée du composant d'IHM qui ont été créés pour un port d'entrée du composant composite, seul un lien est créé entre un port du composant d'IHM et le port du composant composite (les liens internes se chargeant de la consistence des valeurs, un seul lien est nécessaire). Le listing B.46 présente la création de l'ensemble des couples qui correspondent aux liens de données entre entrées.

```
create_data_link_in(_,[],[]).
2 create_data_link_in(PortFc,[PortUi|OtherPortUi],[[PortUi,PortFc]|OtherLink]):-
        create_data_link_in(PortFc,OtherPortUi,OtherLink),!.
```

Listing B.46 – Création des liens de données

B.5.2 Création des ports d'événements

Ce qui suit présente la création des ports d'événement associés à une action. Pour réaliser la création des ports d'événement, on utilise en plus du port d'action et de la liste qui associe au port d'action ses représentations graphiques, les listes des identifiants des ports d'entrées et de sorties associées à cette action. Ces listes sont nécessaires à la création de la définition du port d'action qui correspond au fait *event_data_association* qui permet d'associer à un événement ses entrées et sorties. Le résultat obtenu suite à l'usage du prédicat *create_ui_component_action_port* (cf. listing B.47) est trois listes qui contiennent l'ensemble des définitions des ports d'événement, l'ensemble des liens et l'ensemble des identifiants. La création des ports d'événement est réalisé par le prédicat *create_ui_port_action*.

```
create_ui_component_action_port(PortAction,ListOfUIaction,ListPortIdIn,ListPortIdOut,
2                               ResultingPortAction,ResultindEventLinkOut,
                                ListPortIdAction):-
        find_good_element(PortAction,ListOfUIaction,[_,ElemAction]),
        create_ui_port_action(PortAction,ListPortIdIn,ListPortIdOut,
                              ElemAction,ListPortIdAction,ResultingPortAction),
7       create_event_link_out(PortAction,ListPortIdAction,ResultindEventLinkOut),!.
```

Listing B.47 – Création des ports d'événement et des liens associés

Le listing B.48 présente le prédicat *create_ui_port_action* qui permet la création des ports d'événement. La création se fait par itération sur l'ensemble des représentations graphiques associées à un port d'action. La réalisation du port d'événements se fait de la même manière qu'un port de donnée. On génère dans un premier temps un identifiant unique. On crée ensuite la définition du port d'événement qui contient l'identifiant créé, le nom présent dans la description ainsi que la liste des ports d'entrée et de sortie du composant d'IHM auxquels il est associé. L'identifiant est ajouté à la liste des identifiants et la définition à la liste des définitions.

```
create_ui_port_action(_PortAction,_ListPortIdIn,_ListPortIdOut,[],[],[]).
create_ui_port_action(PortAction,ListPortIdIn,ListPortIdOut,[[_CUI,_ID,Name]|OtherAction],
3                     [PortID|OtherID],ResListPortInfo):-
        gensym('ui_action',PortID),
        TmpPortInfo=[PortID,Name,ListPortIdIn,ListPortIdOut],
        create_ui_port_action(PortAction,ListPortIdIn,ListPortIdOut,
                              OtherAction,OtherID,ListPortInfo),
8       append([TmpPortInfo],ListPortInfo,ResListPortInfo).
```

Listing B.48 – Création de l'ensemble des ports d'événement

La liste des identifiants des ports d'événement créés est utilisée dans le prédicat *create_event_link* pour permettre la création des liens d'événement. Le listing B.49 présente la création des liens d'événement dont le résultat correspond à une liste de couple dont le premier élément est l'identifiant d'un port d'événement du composant d'IHM et le second élément est l'identifiant d'un port d'action du composant fonctionnel.

```
create_event_link(_,[],[]).
create_event_link(PortFc,[PortUi|OtherPortUi],[[PortUi,PortFc]|OtherLink]):-
        create_event_link(PortFc,OtherPortUi,OtherLink),!.
```

Listing B.49 – Création des liens d'événement

B.6 Synthèse sur le moteur de composition

Dans cette annexe, nous vu la réalisation du moteur de composition. Celui-ci a pour objectif de créer l'ensemble des composants d'IHM associés à la composition fonctionnelle qui est représentée par un composant composite. Ces composants d'IHM créés utilisent l'ensemble des informations présentes dans les IHM des composants fonctionnels impliqués dans la composition. Il crée également l'ensemble des liens pour permettre la communication entre les composants d'IHM et la composition fonctionnelle.

Ce moteur de composition détecte également les points de conflits qui sont associés à des points de fusion. Les conflits apparaissent lorsque deux représentations graphiques (ou plus) associées à un point de fusion sont différentes. Une première phase, intégrée directement au sein du moteur de composition, permet de supprimer l'ensemble des confits qui sont inconsistants par rapport aux exigences du port du composant. Ceci est possible pour les ports de données et cela se base sur la cardinalité. Enfin pour l'ensemble des conflits restants, il advient au développeur de les résoudre soit de manière automatique en choisissant une politique de résolution de conflits pour chacun des éléments en conflit, soit de manière manuelle en fournissant lui-même les informations nécessaires pour créer une nouvelle représentation graphique. La résolution des conflits se fait par l'intermédiaire d'une extra-IHM qui permet au développeur d'avoir l'ensemble des données pour permettre de résoudre les conflits.

C

Règles de transformations et méta-modèles eCore

C.1 Abstraction du WSDL dans le méta-modèle AliasComponent

C.1.1 Représentation du méta-modèle de WSDL en eCore

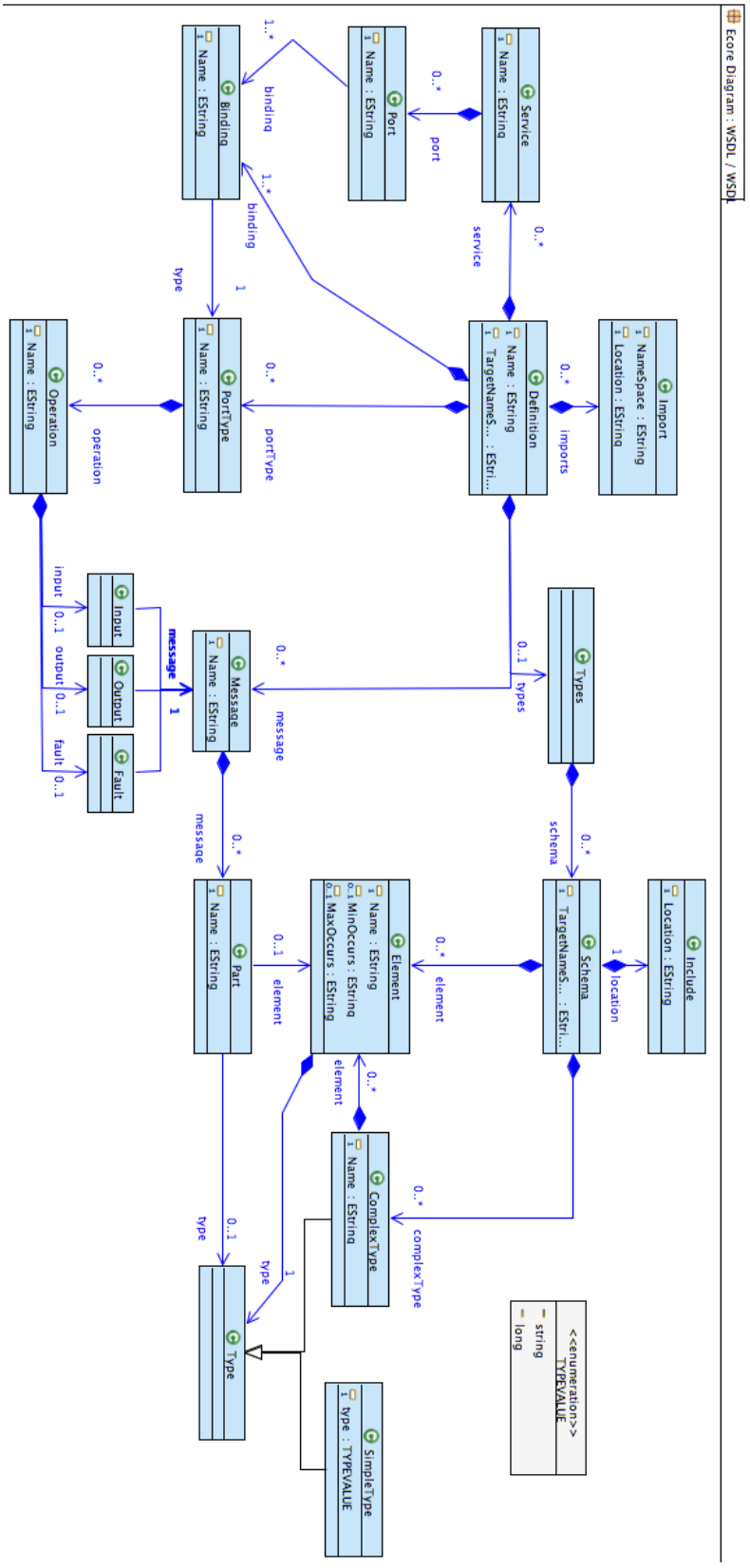


FIGURE C.1 – Méta-modèle eCore de WSDL

C.1.2 Représentation du méta-modèle d'AliasComponent en eCore

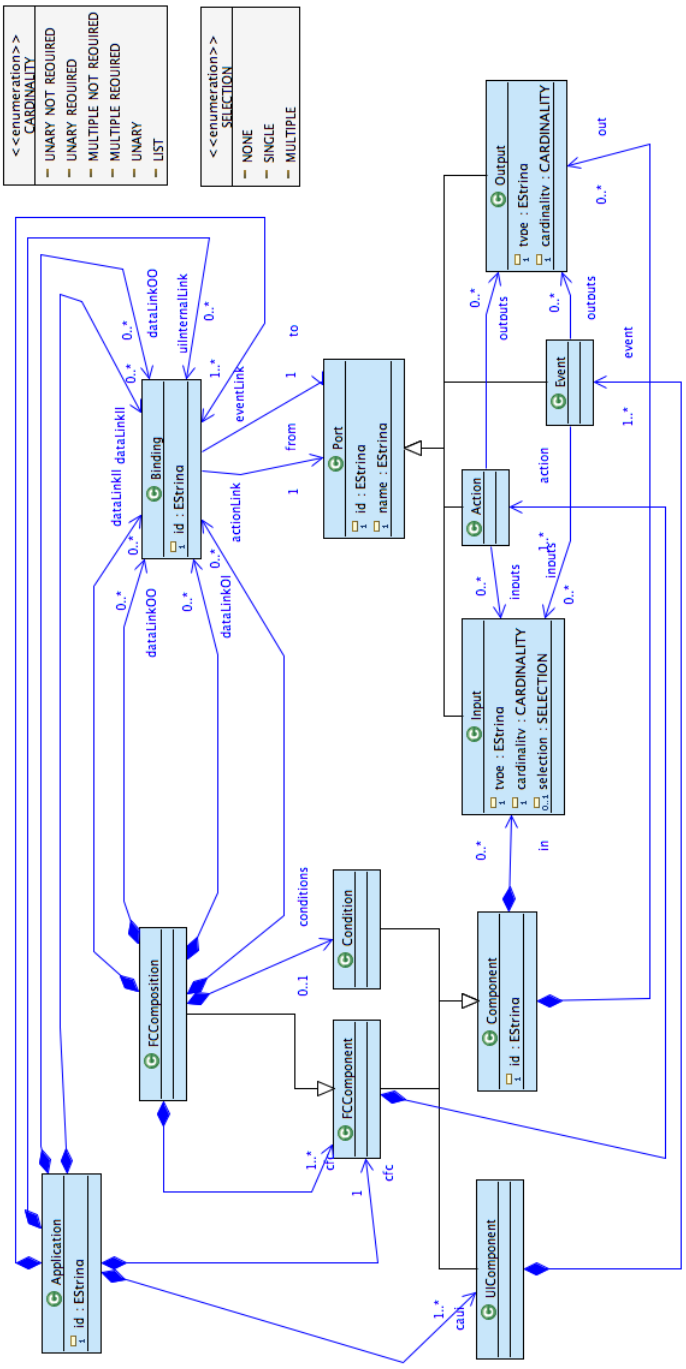


FIGURE C.2 – Méta-modèle eCore d'AliasComponent

C.1.3 Règles de transformation ATL

```
-- @path AliasComponent=/Resources/model/AliasComponent.ecore
-- @path WSDL=/Resources/model/WSDL.ecore

module wsdl2aliascomponent;
create OUT : AliasComponent from IN : WSDL;

helper def: counterA : Integer = 0;
helper def: counterI : Integer = 0;
helper def: counterO : Integer = 0;
helper def: inputP : Sequence(AliasComponent!Input) = Sequence{};
helper def: outputP : Sequence(AliasComponent!Output) = Sequence{};

helper def: mapComplex2Sequence : Map(String,Sequence(WSDL!Element)) =
   WSDL!ComplexType.allInstances()->
      iterate(e; ret : Map(String, Sequence(WSDL!Element)) = Map{}|
        ret->including(e.Name, e.element)
      );

helper def: retrieveSequenceFromComplex (elm : WSDL!ComplexType) : Sequence(WSDL!Element) =
   thisModule.mapComplex2Sequence.get(elm.Name);


rule WSDL2AliasComponent {
   from
      wsdl : WSDL!Definition
   to
      out : AliasComponent!FCComponent(
         id <- wsdl.Name,
        actionPorts <- WSDL!Operation->allInstances()->
           collect(e | thisModule.Operation2Action(e)),
        inputPorts <- thisModule.inputP,
        outputPorts <- thisModule.outputP
      )
}

lazy rule Operation2Action {
   from
      op : WSDL!Operation
   using{
      ct : Integer = thisModule.counterA;
   }
   to
      a : AliasComponent!Action (
        name<-op.Name,
        id<-'action_'+ct,
        inputs <- if op.input.message.message.first().type.oclIsUndefined()
          then if op.input.message.message.first().element.type.oclIsTypeOf(WSDL!ComplexType)
            then thisModule.retrieveSequenceFromComplex(
              op.input.message.message.first().element.type) ->
                collect(e | thisModule.Element2Input(e))
            else thisModule.Element2Input(op.input.message.message.first().element)
            endif
          else if op.input.message.message.first().type.oclIsTypeOf(WSDL!ComplexType)
            then thisModule.retrieveSequenceFromComplex(
              op.input.message.message.first().type) ->
                collect(e | thisModule.Element2Input(e))
            else thisModule.SimpleType2Input(
              op.input.message.message.first().Name,op.input.message.message.first().type)
            endif
          endif,
          outputs <- if op.output.message.message.first().type.oclIsUndefined()
            then if op.output.message.message.first().element.type.oclIsTypeOf(WSDL!ComplexType)
              then thisModule.retrieveSequenceFromComplex(
                op.output.message.message.first().element.type) ->
                  collect(f | thisModule.Element2Output(f))
              else thisModule.Element2Output(op.output.message.message.first().element)
              endif
            else if op.output.message.message.first().type.oclIsTypeOf(WSDL!ComplexType)
              then thisModule.retrieveSequenceFromComplex(
                op.output.message.message.first().type) ->
                  collect(f | thisModule.Element2Output(f))
              else thisModule.SimpleType2Output(
                op.output.message.message.first().Name,op.output.message.message.first().type)
              endif
```

```
            endif)
   do{
77    thisModule.counterA <- thisModule.counterA + 1;
   }
}

lazy rule SimpleType2Input{
82 from
      nameE : String ,
      typeEl : String
   using{
      ct : Integer = thisModule.counterI;
87 }
   to
      input : AliasComponent!Input (
         id <- 'input_'+ct,
               name <- nameE,
92       type <- if typeEl.type = #string then 'string' else 'long' endif,
         cardinality <- #UNARY_REQUIRED)
   do{
      thisModule.counterI <- thisModule.counterI + 1;
      thisModule.inputP->including(input);
97 }
}

lazy rule Element2Input{
   from
102   el : WSDL!Element
   using{
      ct : Integer = thisModule.counterI;
      typeEl : WSDL!SimpleType = el.type;
   }
107 to
      input : AliasComponent!Input (
        id <- 'input_'+ct,
        name <- el.Name,
        type <- if typeEl.type = #string then 'string' else 'long' endif,
112     cardinality <- if el.MaxOccurs = 'unbounded' then
              if el.MinOccurs = '0' then #MULTIPLE_NOT_REQUIRED
              else #MULTIPLE_REQUIRED
              endif
           else if el.MinOccurs = '0' then #UNARY_NOT_REQUIRED
117           else #UNARY_REQUIRED
              endif
           endif)
   do{
      thisModule.counterI <- thisModule.counterI + 1;
122   thisModule.inputP->including(input);
   }
}

lazy rule SimpleType2Output{
127 from
      nameE : String ,
      typeEl : String
   using{
      ct : Integer = thisModule.counterI;
132 }
   to
      output : AliasComponent!Output (
         id <- 'output_'+ct,
               name <- nameE,
137      type <- if typeEl.type = #string then 'string' else 'long' endif,
         cardinality <- #UNARY_NOT_REQUIRED)
   do{
      thisModule.counterO <- thisModule.counterO + 1;
      thisModule.outputP->including(output);
142 }
}

lazy rule Element2Output{
   from
147   el : WSDL!Element
   using{
      ct : Integer = thisModule.counterO;
      typeEl : WSDL!SimpleType = el.type;
```

```
    }
152 to
        output : AliasComponent!Output (
          id <- 'output_'+ct,
          name <- el.Name,
          type <- if typeEl.type = #string then 'string' else 'long' endif,
157       cardinality <- if el.MaxOccurs = 'unbounded' then
                 if el.MinOccurs = '0' then #MULTIPLE_NOT_REQUIRED
                 else #MULTIPLE_REQUIRED
                 endif
              else if el.MinOccurs = '0' then #UNARY_NOT_REQUIRED
162               else #UNARY_REQUIRED
                  endif
              endif)
    do{
        thisModule.counterO <- thisModule.counterO + 1;
167     thisModule.outputP->including(output);
    }
}
```

Listing C.1 – Règles de transformation de modèles pour l'abstraction

Résumé

La réutilisation d'applications n'est pas un problème récent et de nombreux travaux ont étudié différentes façons de réutiliser tout ou partie d'une application. L'apparition des services et des composants au niveau fonctionnel facilite la construction et l'extension d'applications par composition de fonctionnalités indépendantes. En parallèle, des travaux sur les Interfaces Homme–Machine (IHM) proposent des solutions pour créer des IHM par composition d'IHM existantes. Mais ce n'est que récemment que des travaux se sont intéressés à composer l'ensemble d'une application. Malgré cela, ils ne proposent que des compositions simples tant au niveau des IHM que fonctionnel.

Ces travaux de thèse proposent une approche pour faciliter le travail du développeur en lui permettant de réutiliser l'intégralité des applications à composer, c'est-à-dire, la partie fonctionnelle, l'IHM et les interactions entre l'IHM et la partie fonctionnelle. La proposition faite dans cette thèse lui permet de continuer d'utiliser les outils industriels dont il dispose pour composer les parties fonctionnelles des applications. Ainsi, c'est au travers de cette composition fonctionnelle ainsi que des interactions entre la partie fonctionnelle et l'IHM présentes dans les applications que se fera l'identification des éléments d'IHM à conserver et la création de l'application résultante.

Cette approche repose sur un méta-modèle `AliasComponent` qui fournit une architecture à base de composants permettant de décrire à la fois les applications (en distinguant la partie fonctionnelle, les IHM et les interactions entre les deux) et la composition fonctionnelle. Ce méta-modèle permet ainsi d'extraire des applications existantes et de la composition fonctionnelle l'ensemble des informations essentielles à la réalisation de la composition des applications. Elle s'appuie aussi sur une formalisation qui permet la réalisation de la composition, c'est-à-dire, la déduction des éléments d'IHM à conserver, la détection de doublons non identiques (lorsque plusieurs widgets différents peuvent convenir), l'aide à la résolution de ceux-ci et la création de l'application résultante de la composition. Cette formalisation pose également les propriétés que doivent respecter les applications et le résultat de la composition assurant ainsi une application résultante fonctionnelle. Le méta-modèle et la formalisation sont intégrés au sein d'un processus d'aide à la composition. La solution proposée a, quant à elle, été mise en œuvre au sein d'un atelier d'aide à la composition qui a permis de valider l'approche.

Mots-clés : composition d'applications, réutilisation d'applications, détection/résolution de conflits, formalisation, ingénierie dirigée par les modèles

Abstract

Composition of Applications and their User Interfaces driven by the Functional Composition

Software reuse is not a recent issue. Many approaches have been proposed in the literature enabling the reuse of part or complete application including its functionalities and User Interface (UI). At the functional level, based on the appearance of services and components, these approaches ease the process of building and extending applications by composing independent functionalities. At the UI level, they offer solution to create new UI by the composition of existing ones. Several approaches focus on the application composition. However, they only propose to realize simple composition.

Also focusing on the application composition, the approach presented in this thesis eases better the whole composition process. It allows software developer to design new application by reusing all components of existing applications including functionalities, UIs and interaction between functionalities and UIs. Our approach enables software developer to use existing industrial tools serving for functionality composition. Thus, through this process, desirable widgets can be identified and reused to create the resulting application.

The proposed approach is based on a meta-model called `AliasComponent` that provides a component-based architecture to describe applications by distinguishing the functional part, UIs and interactions between them, and functional composition. This metamodel focuses only on important information captured from existing applications, and functional composition. A formalization has been described to validate the intermediate results during the composition process. With this formalization, any useful widgets are selected, in case of multiple widgets, a list of possible ones is show, helping the developer to make a choice and finally create the resulting application. In this formalization, we define some properties that application and composition result must respect and ensure that the resulting application is useful. The metamodel and formalization are integrated in a composition tool which validites the approach presented in this thesis.

Keywords : applications composition, software reuse, conflict detection/resolution, formalization, model driven engineering