# Interprétations de la composition d'activités

Mireille Blay-Fornarino

▶ To cite this version:

Mireille Blay-Fornarino. Interprétations de la composition d'activités. Informatique [cs]. Université Nice Sophia Antipolis, 2009. tel-00460768

HAL Id: tel-00460768
https://theses.hal.science/tel-00460768

Submitted on 2 Mar 2010

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Habilitation à diriger des recherches

École doctorale « Sciences et Technologies de l'Information
et de la Communication » de Nice - Sophia Antipolis
Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES ET TECHNIQUES

---

## INTERPRÉTATIONS DE LA COMPOSITION D'ACTIVITÉS

---

*présentée et soutenue publiquement par*

**Mireille Blay-Fornarino**

*Soutenue le* 16 avril 2009 *à* l'Ecole Polytechnique Universitaire *devant le jury composé de*

*Président du Jury*

Michel RIVEILL — Université de Nice - Sophia Antipolis

*Rapporteurs*

Laurence DUCHIEN — Université de Lille - LIFL - INRIA
Jean-Marc JÉZÉQUEL — Université de Rennes I - IRISA
Viviane JONCKERS — Vrije Universiteit Brussel, VUB, Belgique

*Examinateurs*

Yves CASEAU — DGA Bouygues Telecom
Paul FRANCHI — Université de Nice - Sophia Antipolis

*À ma Famille,*
*mes amies Anne-Marie et Rose.*

*— Mireille.*

# TABLE DES MATIÈRES

**1 Introduction** **3**
1.1 Vocabulaire : abstraction, interprétation, activité, composition. . . . . . . . . 4
1.2 Contexte général de nos travaux . . . . . . . . . . . . . . . . . . . . . . . . 5
1.2.1 Assemblages logiciels . . . . . . . . . . . . . . . . . . . . . . . . . . 6
1.2.1.1 Éléments d'assemblage et comportement . . . . . . . . . . . 6
1.2.1.2 Assemblage de composants et validation . . . . . . . . . . . 7
1.2.1.3 Infrastructure et indépendance plate-forme . . . . . . . . . 8
1.2.2 Activités et Workflow . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.2.2.1 Parallélisme d'une application . . . . . . . . . . . . . . . . . 9
1.2.2.2 Langages de description d'un workflow et Abstraction . . . 10
1.2.2.3 Diagrammes de comportements en UML . . . . . . . . . . . 11
1.2.3 Séparation des préoccupations . . . . . . . . . . . . . . . . . . . . . 12
1.2.3.1 Programmation par Aspects et Interactions . . . . . . . . . 13
1.2.4 Compositions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
1.2.4.1 Composition de workflows . . . . . . . . . . . . . . . . . . . 15
1.2.4.2 Composition de modèles . . . . . . . . . . . . . . . . . . . . 16
1.2.5 Langages et métacompilation . . . . . . . . . . . . . . . . . . . . . . 19
1.2.6 En résumé . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
1.3 Démarche . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
1.3.1 Abstraction et Interprétation . . . . . . . . . . . . . . . . . . . . . . 21
1.3.2 Compositions d'activités : un point de vue transformation de modèles 21
1.3.3 Quelques propriétés de l'opération de composition . . . . . . . . . . . 23
1.4 Rappel du plan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**I Modélisation de la composition d'activités** **25**

**Dans cette partie...** **27**

**2 Modélisation et formalisation des activités** **29**
2.1 Exemple : Domaine fil rouge . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.2 Modélisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2.2.1 Simplifications vis-à-vis d'UML . . . . . . . . . . . . . . . . . . . . . . . 31
2.2.2 Sémantique d'exécution d'une activité composite . . . . . . . . . . . . 31
2.2.3 Domaines d'application . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.3 Formalisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.3.1 Domaines d'application : domaines d'interprétation . . . . . . . . . . 34
2.3.2 Action . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.3.3 Activité simple . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.3.4 Activité composite . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
2.3.4.1 Relation de précédence . . . . . . . . . . . . . . . . . . . . . . . 38
2.3.4.2 Relation de condition . . . . . . . . . . . . . . . . . . . . . . . . 38
2.3.5 Domaine d'application : contraintes . . . . . . . . . . . . . . . . . . . 40
2.3.6 Système . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.3.7 Substitutions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
2.3.8 Equivalences au sein d'un système . . . . . . . . . . . . . . . . . . . . 42
2.3.9 Normalisation d'une activité composite . . . . . . . . . . . . . . . . . 44
2.3.10 Normalisation sur domaine . . . . . . . . . . . . . . . . . . . . . . . . 45

**3 Composition d'activités 47**
3.1 Vue synthétique de la composition d'activités composites . . . . . . . . . . . 47
3.2 Activités pivots et ensembles de fusion . . . . . . . . . . . . . . . . . . . . . 49
3.3 Transformations et compositions des transformations . . . . . . . . . . . . . 53
3.4 Fusion d'activités simples au sein d'un ensemble d'activités . . . . . . . . . 55
3.5 Normalisation partielle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
3.6 Composition d'un ensemble d'activités composites . . . . . . . . . . . . . . . 59

**4 Propriétés des compositions d'activités 63**
4.1 Idempotence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.2 Préservation des propriétés . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3 Absence de duplication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.4 Indépendance de l'ordre des activités . . . . . . . . . . . . . . . . . . . . . 67
4.5 Associativité . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

**5 Conclusion : Eléments pour la composition d'activités 71**
5.1 Modélisation et domaine d'application . . . . . . . . . . . . . . . . . . . . . 71
5.1.1 Mise en conformité : limites et perspectives . . . . . . . . . . . . . . . 71
5.2 Composition et Interprétation . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2.1 Détection des paires de fusion et construction des ensembles de fusion 72
5.2.2 Transformations et composition des transformations . . . . . . . . . 72
5.2.3 Fusion d'activités simples . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.2.4 Composition d'activités composites . . . . . . . . . . . . . . . . . . . . 73
5.3 Propriétés . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.3.1 Idempotence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.3.2 Préservation des propriétés . . . . . . . . . . . . . . . . . . . . . . . . 74
5.3.3 Absence de duplication . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.3.4 Indépendance de l'ordre des activités . . . . . . . . . . . . . . . . . . . 74
5.4 Suite du document . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

**II Applications** **75**

**Dans cette partie...** **77**

**6 Compositions d'Interactions entre composants hétérogènes** **79**
6.1 Domaine applicatif et objectifs . . . 80
6.1.1 Des adaptations de composants aux interactions logicielles . . . 81
6.1.2 Exemples de compositions d'interactions . . . 83
6.1.3 Notre approche : une modélisation exécutable des interactions . . . 85
6.1.3.1 Service d'interactions . . . 86
6.1.3.2 Des interactions aux compositions d'activités . . . 86
6.2 Formalisation et composition dans le domaine des interactions . . . 86
6.2.1 Domaine applicatif et contraintes sur domaine . . . 87
6.2.1.1 Variables . . . 87
6.2.1.2 Informations . . . 87
6.2.1.3 Contraintes sur domaine . . . 87
6.2.2 Détection des ensembles de fusion . . . 88
6.2.2.1 Activités pivots . . . 88
6.2.2.2 Détection des paires de fusion . . . 88
6.2.2.3 Propriétés . . . 89
6.2.3 Transformations et composition de transformations . . . 90
6.2.3.1 Propriétés de composition des transformations . . . 90
6.2.4 Fusion d'activités simples . . . 90
6.2.4.1 Propriétés de la fusion d'activités simples . . . 92
6.2.5 Normalisation partielle . . . 93
6.2.6 Normalisation sur domaine . . . 93
6.2.7 Composition d'un ensemble d'activités composites . . . 93
6.2.7.1 Traitements des ensembles de fusion . . . 93
6.2.7.2 Pas de réintroduction de paires de fusion . . . 93
6.2.7.3 Respect des propriétés exigées . . . 93
6.2.7.4 Remarque . . . 94
6.2.8 Propriétés des compositions d'interactions . . . 94
6.2.8.1 Idempotence . . . 94
6.2.8.2 Préservation "partielle" des propriétés . . . 94
6.2.8.3 Non Duplication par la fonction de composition des interactions . . . 94
6.2.8.4 Indépendance des ordres de composition . . . 95
6.2.8.5 Associativité . . . 95
6.3 Formalisation des schémas et des règles d'interactions . . . 96
6.3.1 Modélisation . . . 97
6.3.1.1 Objets et Opérations . . . 97
6.3.1.2 Règles et Interactions . . . 97
6.3.1.3 Schémas . . . 98
6.3.1.4 Configuration . . . 99
6.3.2 Pose et compositions d'interactions . . . 99
6.4 Mise en œuvre : le service d'interactions . . . 101
6.4.1 Interactions à l'exécution . . . 101
6.4.2 Composant interagissant . . . 102

6.4.2.1 Interopérabilité et interactions . . . . . 102
6.4.2.2 Mises en œuvre . . . . . 102
6.4.3 Serveur d'interactions . . . . . 103
6.4.3.1 Pose, composition et destruction d'interactions . . . . . 103
6.4.3.2 Navigation et Analyse du graphe d'interactions . . . . . 103
6.4.4 Langage de Spécifications des Interactions (ISL) . . . . . 103
6.4.5 Environnement de programmation . . . . . 104
6.5 Conclusion et perspectives . . . . . 104
6.5.1 Retours sur le service d'interactions et expérimentation . . . . . 105
6.5.2 Interactions et programmation par aspects . . . . . 106
6.5.3 Perspectives . . . . . 108

**7 Composition d'orchestrations 111**
7.1 Domaine applicatif et objectifs . . . . . 112
7.1.1 Enjeux de la composition d'orchestrations . . . . . 112
7.1.2 Exemple de composition d'orchestrations . . . . . 114
7.1.3 Notre approche : composition par reconnaissance des activités surchargées . . . . . 115
7.2 Formalisation et composition dans le domaine des orchestrations . . . . . 116
7.2.1 Domaine applicatif et contraintes sur domaine . . . . . 116
7.2.1.1 Variables . . . . . 117
7.2.1.2 Informations . . . . . 117
7.2.1.3 Contraintes sur domaine . . . . . 118
7.2.2 Détection des ensembles de fusion . . . . . 118
7.2.2.1 Activités pivots . . . . . 118
7.2.2.2 Détection des paires de fusion . . . . . 119
7.2.2.3 Propriétés . . . . . 119
7.2.3 Transformations et composition de transformations . . . . . 119
7.2.3.1 Composition . . . . . 120
7.2.3.2 Propriété de la composition des transformations . . . . . 121
7.2.4 Fusion d'activités simples . . . . . 121
7.2.4.1 $receive^n$ . . . . . 122
7.2.4.2 $invocation^n$ . . . . . 122
7.2.4.3 $reply^n$ . . . . . 125
7.2.4.4 Propriétés de la fusion d'activités simples . . . . . 125
7.2.5 Normalisation partielle . . . . . 126
7.2.6 Normalisation sur domaine . . . . . 127
7.2.7 Composition d'un ensemble d'orchestrations . . . . . 127
7.2.7.1 Traitements des ensembles de fusion . . . . . 127
7.2.7.2 Respect des propriétés exigées . . . . . 127
7.2.8 Propriétés des compositions d'orchestrations . . . . . 127
7.2.8.1 idempotence . . . . . 127
7.2.8.2 Préservation "partielle" des propriétés . . . . . 128
7.2.8.3 Circonscription de l'absence de duplication . . . . . 128
7.2.8.4 Indépendance des ordres de composition, pas des choix utilisateurs . . . . . 128
7.2.8.5 Pas d'associativité . . . . . 129
7.3 Mise en œuvre et applications . . . . . 129

7.4 Conclusion et Perspectives . . . 130
7.4.1 Retour sur la composition d'orchestrations . . . 130
7.4.2 Perspectives . . . 131
7.4.2.1 Modélisation et composition étendue des orchestrations . . . 131
7.4.2.2 Atelier de composition et passage à l'échelle . . . 131
7.4.2.3 Flots de données et orchestrations . . . 132
7.4.2.4 Traçabilité . . . 132

**III Conclusion et perspectives 133**

**8 Contributions 135**

**9 Perspectives 137**
9.1 Représentation intermédiaire . . . 137
9.1.1 Perspective prédominante : représentation intermédiaire incluant la gestion des flux de données . . . 139
9.2 Détermination des ensembles de fusion . . . 139
9.2.1 Perspective souhaitée : application à la composition de scénarios . . . 140
9.3 Validité des compositions d'activités . . . 140
9.3.1 Perspective prédominante : Associer contrats et composition . . . 140
9.4 Transformations . . . 140
9.4.1 Perspective prédominante : Composition de transformations, idempotence et rejeux . . . 141
9.5 Auto-organisation . . . 141

**A Liste sélective de résultats et d'encadrements 143**
A.1 Modélisation de la composition d'activités . . . 143
A.2 Composition d'interactions . . . 144
A.3 Composition d'orchestrations . . . 146

**Bibliographie 154**

# Remerciements

Comme tout travail de recherche, celui-ci n'est pas issu d'un seul esprit mais de la synergie entre les personnes rencontrées au fil des réunions et conférences et ceux qui partagent notre quotidien. Donc au delà des remerciements que je m'apprête à adresser à ceux qui ont contribué de manière évidente à l'aboutissement de ce mémoire, je rappellerai les contributions de chacun en début de chaque chapitre important, en essayant de n'oublier personne.

Ma pensée va d'abord à Rose qui a toujours su aller au delà des mots pour rechercher l'essentiel dans chacun de nous. J'aurais tant voulu en présentant cette habilitation lui dire merci de vive voix pour toutes ces fois où sa confiance m'a permis de croire que l'on pouvait changer le monde simplement en défendant honnêtement ses idées. Scientifiquement et humainement je lui dois tant !

Mon amie, Anne-Marie, est assurément celle à qui je dois d'être arrivée au bout de ce mémoire. Chaque jour, elle est présente aussi bien dans la recherche, l'enseignement, l'administration, ... C'est mon amie.

Mes remerciements vont ensuite à Michel Riveill qui m'a si souvent permis de concrétiser mes idées et mes projets. Sa venue dans notre tout jeune projet Rainbow a vraiment été une source d'informations, de remises en question et d'accomplissements. Il a su faire de notre projet une équipe. Grâce à cette équipe, j'ai pu m'enthousiasmer pour de nouveaux espaces de recherche. Depuis quelques mois, nous avons construit une nouvelle équipe, Modalis, dont chacun des membres à tour à tour influencé un peu de la rédaction de ce mémoire d'habilitation.

Je remercie également Laurence Duchien, Jean-Marc Jezequel et Viviane Jonckers pour avoir accepté de lire ce long mémoire et d'être les rapporteurs de mon habilitation.

Paul Franchi-Zannetacchi a eu un impact précieux sur mes recherches. Je le remercie pour ses multiples conseils, son regard si différent sur l'"informatique" au fil de toutes ces années et pour avoir finalement accepté d'être dans ce jury.

Yves Caseau est intervenu à plusieurs reprises dans ma vie de chercheur, en particulier lors de la création du projet Rainbow où, en tant qu'évaluateur, il a soutenu la création du projet. Puis quelques années plus tard, il fit le rapprochement entre nos travaux et les workflows. Ces interventions ont influencé mes recherches. Aujourd'hui il a accepté de participer à ce jury et j'en suis très fière.

Et puis, je suis reconnaissante aux "thésards", Stéphane, Pascal, Olivier, Clémentine, Sébastien, avec qui l'essence de ce manuscrit s'est construite au fil des années, au travers des rires, des confrontations et des "brainstormings" si enrichissants.

Bien sûr, il y a beaucoup d'autres personnes que je dois remercier, qu'ils me pardonnent de ne pas les citer plus longuement ici : Audrey O., Claudine P., Christian Q., David E., Eric M., Diane L., Jean-Yves T., Jean-Marc J., Johan M., Laurent H., Laurence D., Philou, Philippe L., Sabine M., Stéphane L. ....

Enfin, puisque peu d'occasions nous sont données de dire merci à nos proches, je profite de celle-ci pour remercier Claude, nos enfants, et ma grande famille...

*Merci.*
— Mireille.

# Génèse : en guise d'avant-propos

Ce document est le reflet d'une quête dont l'origine remonte à Gilles Kahn [1] qui nous dit : "Si en logique on peut exprimer des dépendances entre des termes, pourquoi ne les exprime-t-on pas entre des objets ?".

**Objets, logique et dépendance** Mais que signifie une dépendance entre des objets ? Non, ce n'est pas seulement un attribut, une référence. Dépendance veut dire relation de subordination, de corrélation, collaborations. Une dépendance n'est pas liée à la classe, les objets issus d'une même classe sont en interaction différemment avec les autres objets de leur environnement. *No object is an island.* [BC89].

$\alpha$ →Nous avons interprété alors les dépendances comme des entités de première classe "interactions entre des objets".

**Interactions et compositions** Les interactions s'expriment séparément. Les objets sont impliqués dans différentes interactions. Il faut donc composer les interactions. Mais s'il y a vraiment une expression séparée des interactions, alors nous ne pouvons pas attendre de l'utilisateur qu'il ordonne les interactions !

$\beta$ →Nous imaginons un procédé, dit de fusion, qui assure la composition commutative et associative des interactions.

**Modèle d'interactions et composants** Les objets sont devenus des composants. Et la communauté "objet" a enfin reconnu l'existence des liaisons comme entités de première classe. Les dépendances en terme de fournisseurs et consommateurs sont ainsi clairement identifiées. Les interactions s'identifient-elles alors à de multiples liaisons ? Non, elles ne sont pas anticipées, elles s'interposent dynamiquement. Les composants ne sont pas tous implémentés dans le même espace.

$\gamma$ → Un modèle d'interaction est conçu. Les interactions s'"expriment" indépendamment des plate-formes et sont mises en œuvre à l'exécution par transformation vers de

1. dont je trahis sûrement un peu les propos.

multiples plate-formes. Chaque composant interprète sa partie des interactions. Le service d'interactions est né.

**Diversité des compositions** Les interactions s'appliquent à différents contextes. Mais il semble qu'il faille adapter les compositions pour répondre à la spécificité des domaines. Comment assurer que les propriétés d'associativité et de commutativité sont toujours vérifiées dans ces nouveaux domaines ? Elles ne sont pas forcément voulues.

$\delta \rightarrow$ Les interactions deviennent aspects d'assemblages, fonctions génériques, contrôles dans les frameworks, ....

**Activités et Fusions** Le contexte évolue encore. Les composants ne suffisent plus. Interopérabilité, agilité, intégration, adaptation autonomique deviennent des défis qu'il est urgent d'aborder. Un presque nouveau paradigme apparaît : les services et leurs orchestrations. Ces dernières sont l'expression d'interactions. Elles contrôlent les compositions. Tout en elles est activités ; elles sont des activités. C'est elles que l'on veut composer, fusionner.

$\varepsilon \rightarrow$Nous nous intéressons alors à la fusion des orchestrations. Le procédé change. Nous ne considérons plus les messages comme clef des dépendances mais l'orchestration et ses activités. Les premiers algorithmes de composition d'orchestrations sont établis.

**Interprétations de la composition d'activités** Quel est le dénominateur commun à ces travaux ? Ce n'est pas les interactions, nous en avons des interprétations trop différentes. Les compositions, elles-aussi nous les envisageons différemment. Pourtant, au fil des applications, des éléments clefs reviennent. Une intuition ou plutôt une hypothèse, il existe un modèle et un algorithme commun sous-jacent qui sous-tend à ces travaux. Le regard se déplace. Les activités deviennent le centre d'intérêt tandis que la composition devient un algorithme (interprétation libre) et les dépendances avec les domaines applicatifs des interprétations.

$\zeta \rightarrow$Ce fut la rédaction longue et laborieuse de ce mémoire.

Mais il n'y eut pas de repos.

# CHAPITRE 1

# Introduction

*"Extensible systems are in principle modular, have no final form or final integration phase, cannot be subjected to final total analysis, cannot be exhaustively tested, and have to allow for mutual independence of extension providers" [Szy96].*

Nos travaux de recherche portent sur la composition fiable d'activités logicielles. Ils s'inscrivent dans le cadre général du génie logiciel, et concernent les étapes de conception, implantation et adaptations d'applications logicielles réparties. Le thème central développé est la modélisation des activités et leur composition tout en garantissant différentes propriétés. La spécificité de ce travail et son originalité est d'avoir suivi une approche de la composition non pas dirigée par les langages mais par les éléments clefs de la composition indépendamment de mises en œuvres spécifiques. En fondant ce travail sur une approche formelle, nous proposons une vision unifiée d'un procédé de composition, et caractérisons les différences en terme d'interprétations. En étayant cette formalisation par plusieurs mises en œuvre à la fois en terme d'environnement de programmation et d'applications nous ancrons cette approche dans une réalité fonctionnelle.

## Plan du document

Ce manuscrit est donc composé de quatre parties.
La première, l'introduction (chapitre courant) définit le contexte de ce travail, ses objectifs et la démarche choisie.

La deuxième partie présente notre modélisation des activités (cf. §2), l'algorithme de composition (cf. §3) et les propriétés attendues des compositions (cf. §4), avant de conclure en résumant les éléments nécessaires à la métacomposition (cf. §5).

La troisième partie présente deux applications qui étayent la modélisation et l'algorithme de composition. La première est relative à la composition d'interactions entre composants hétérogènes (cf. §6), la seconde porte sur la composition d'orchestrations de services (cf. §7).

Enfin une dernière partie conclut ce travail par nos perspectives.

## 1.1 Vocabulaire : abstraction, interprétation, activité, composition...

**Abstraction :** *opération intellectuelle, spontanée ou systématique, qui consiste à abstraire. C'est une simplification, en présence de l'objet concret infiniment complexe et perpétuellement changeant, simplification qui nous est imposée, soit par les nécessités de l'action, soit par les exigences de l'entendement, et qui consiste à considérer un élément de l'objet comme isolé, alors que rien n'est isolable, et comme constant alors que rien n'est au repos. ALAIN, Définitions, [Les Arts et les dieux], Paris, Gallimard, 1961 [1951], p. 1028.*

Nous situons l'abstraction dans une démarche ingénierie des modèles : construction d'un modèle.

**Interpréter :** donner un sens à, expliquer, traduire (ex. interpréter un règlement).
Nous parlerons d'interprétation en lui attribuant un sens équivalent aux interprétations sur domaine définies dans le domaine des langages de programmation et plus particulièrement les grammaires attribuées [FZ82].

**Activité :** action d'une personne, d'une entreprise, .. dans un domaine défini, domaine dans lequel s'exerce cette action.(Petit larousse)

Plus précisément nous nous intéressons aux activités définies dans le domaine du logiciel en général, et les actions sont exercées par le logiciel.

**Composition :** Action de former une application par assemblage ou combinaison de plusieurs éléments.
A cette définition informelle sous-tend l'existence de deux formes de composition, l'assemblage et la combinaison. Il s'en suit différentes interprétations du terme composition selon les approches. Nous traitons les deux formes de compositions dans ce document.

**Assemblage :** action permettant de mettre ensemble des éléments pour former un tout cohérent.

L'assemblage ne comporte donc pas de modification des participants. Dans notre contexte, les participants sont des activités.

**Combinaison/Fusion :**
- *Combinaison* : Union, dans des proportions définies, de deux ou plusieurs éléments donnant un nouvel élément ayant des propriétés différentes de celles de ses composants [1].
- *Fusion* Au fig. Combinaison, mélange intime de plusieurs éléments.

Nous faisons le choix du terme "fusion", lorsque à partir de plusieurs activités nous en créons de nouvelles qui n'exercent pas nécessairement les mêmes actions que celles initiales. Cette définition s'oppose à la fusion telle que définie par le "merge" UML et

1. Cette définition est une adaptation de la définition suivante relative à la chimie : Union, dans des proportions définies, de deux ou plusieurs corps donnant un nouveau corps ayant des propriétés différentes de celles de ses composants.

l'opération "Merge" dans [BBF+06] ; elle correspond par contre à l'opération "compose" dans ce dernier. Notre choix s'explique par le sens étymologique de ces mots.

## 1.2 Contexte général de nos travaux

> *Démêler les lignes d'un dispositif, dans chaque cas, c'est dresser une carte, cartographier, arpenter des terres inconnues, c'est ce qu'on appelle un "travail de terrain". Gilles Deleuze*

*Certains passages de cette contextualisation correspondent à l'état de l'art qui a été rédigé dans le cadre du projet Faros et plus spécifiquement à la partie "développement par composition" [BFCD+06], dont j'étais le coordinateur mais auxquels plusieurs autres personnes ont collaboré : Pierre Combes, Laurence Duchien, Tristan Glatard, Philippe Lahire, Stéphane Lavirotte, Clémentine Nemo, Audrey Occello, Renaud Pawlak, Anne-Marie Pinna-Dery, Lionel Seinturier, Jean-Yves Tigli.*

Assembler du logiciel est une pratique courante. Les principes initiaux étaient la division d'un programme en sous-parties ensuite assemblées pour réduire la complexité du développement. La constitution de grandes équipes de développement et la répartition géographique de ses membres ont renforcé les besoins en terme de spécifications détaillées des parties à assembler et de mise en place d'outils adéquats. Aujourd'hui les impératifs de réutilisation et d'adaptation non anticipée au contexte conduisent à étendre les principes initiaux (i) en considérant les sous-parties comme des entités réutilisables, extensibles et adaptables auxquelles il convient donc d'associer des informations sémantiques, et (ii) en intégrant des mécanismes d'adaptation des assemblages qui répondent aux besoins d'évolution des applications [Szy96].

Depuis plusieurs années, l'ingénierie du logiciel s'oriente vers le développement par composition de composants et de services qui autorise une décomposition des fonctionnalités d'une application en entités plus petites et si possibles indépendantes. Les langages de définitions d'architectures et les langages de définition de workflows complètent ces approches en explicitant la manière dont ces entités sont assemblées pour construire des applications de plus grandes tailles, le procédé étant réitéré à plusieurs niveaux si nécessaire. Nous constatons la même évolution à toutes les étapes du développement. Ainsi avec la généralisation de l'ingénierie des modèles, la composition des modèles devient un des principaux domaines de recherche. Ainsi la programmation par "composition" est un élément clef pour répondre aux problèmes de répartition et d'hétérogénéité des composants logiciels, pour décroître la complexité de développement des applications et pour favoriser la gestion des aspects techniques. Sous le terme de programmation par composition, nous entendons donc à la fois programmation séparée des entités de base et composition de ces entités. La composition peut alors consister à orchestrer (e.g. [2] orchestration de services web), à connecter (e.g. construction d'assemblages de composants), à aggréger (e.g. construction d'un composant composite) ou à tisser (e.g. un programme et des aspects sont tissés).

Dans [Szy96], l'auteur souligne déjà combien il est nécessaire de faire évoluer les démarches de développements pour rendre les systèmes vraiment extensibles ce qui in-

2. exempli gratia en latin, utilisée fréquemment en anglais, et tellement plus synthétique que "par exemple" !

clut (i) de prendre en charge des capacités d'extensions séparées des codes, (ii) de tenir compte de la nécessité de travailler dans un monde ouvert, (iii) d'associer plus d'informations sémantiques aux entités à composer. Nous parcourons rapidement les solutions technologiques actuelles pour répondre à ces différents points.

La figure 1.1 visualise le contexte dans lequel se situe ce travail ancré dans le domaine des assemblages logiciels, procédant par une modélisation des workflows à composer et fondé sur une démarche dirigée par les modèles pour l'expression et la mise en oeuvre des compositions.
Les assemblages logiciels constituent les bases des applications que nous ciblons (cf. 1.2.1) et plus particulièrement leur compréhension sous la forme d'un workflow (cf. 1.2.2). La séparation des préoccupations intervient dans la définition des assemblages et des workflows (cf. 1.2.3). Consécutivement à la séparation des préoccupations, la composition va permettre d'assembler ou fusionner les éléments définis séparément au niveau des workflow et des modèles (cf. 1.2.4). Enfin la démarche globale d'abstraction, de raisonnement et d'adéquation à un domaine correspond à une démarche de métacompilation que nous nommons métacomposition (cf. 1.2.5). Nous ne prenons pas en compte dans notre étude les données partagées non explicites.

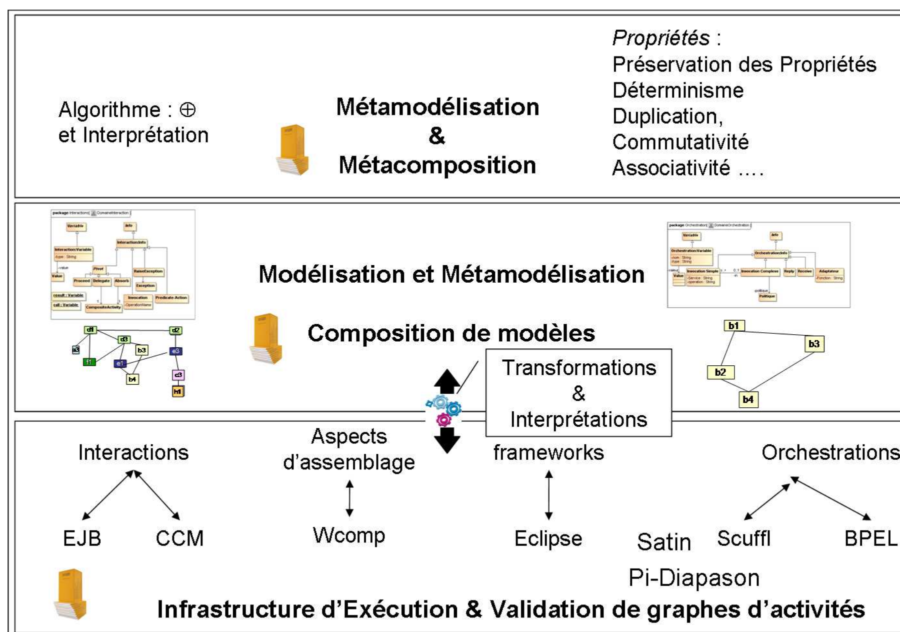



FIGURE 1.1: CONTEXTE DE LA COMPOSITION DES ACTIVITÉS

### 1.2.1 Assemblages logiciels

Le développement par composition de composants et de services représente une solution potentielle à la construction de logiciels de grandes tailles par une décomposition des fonctionnalités en entités plus petites et si possible indépendantes qu'il s'agit alors d'assembler.

#### 1.2.1.1 Éléments d'assemblage et comportement

Nous constatons que pour permettre une vraie réutilisation des composants ou des services, il est nécessaire de les définir par des interfaces mais que celles-ci ne peuvent pas être limitées à définir syntaxiquement le nom, voir la signature des fonctionnalités offertes et requises. Ainsi aux composants sont de plus en plus fréquemment associées les notions d'interface étendue et de contrats [BJPW99, BCL+04]. Celles-ci permettent de préciser les conditions d'usage d'un composant [OPD04, DUVH06, RRB02, TFS04], voire de vérifier des propriétés fonctionnelles et extra fonctionnelles [CRCR05]. Avec la thèse de Pascal Rapicault sous ma co-direction, nous nous sommes plus particulièrement intéressés à exprimer les protocoles d'usages des composants dans le cadre des frameworks afin de contrôler l'utilisation du framework par l'utilisateur [Rap02].

De même, en intensifiant l'usage des services et en renforçant la dynamicité des architectures orientées services, les besoins d'explicitation des engagements des services en terme de qualités garanties se sont accrues. Différents travaux y répondent par la définition de normes explicitant des contrats (WSLA, WSDL, Ws-Agreement,...) [BFCL+07, chapitre 3], par l'adaptation d'outils existants à la contractualisation des services [CRCR05], ou les techniques de spécification comportementale à base d'algèbres de processus [FUMK03].

↪ Ainsi un composant ou un service est porteur d'activités. Son comportement est de plus en plus souvent exposé via des langages adaptés, rendant compte de flots d'activités.

#### 1.2.1.2 Assemblage de composants et validation

Créer un assemblage de composants consiste à "connecter" par des liaisons les composants. La définition explicite des liaisons entre les composants est un des principaux avantages des modèles à composants sur les modèles à objets. Certains travaux cherchent aujourd'hui à assurer que seules des communications explicites entre des composants sont utilisées afin de valider l'intégrité des communications [LCL06, ACN02]. Les liaisons jouent ainsi un rôle important en particulier lorsque les composants sont distants, voir définis dans des plates-formes hétérogènes. Pour accroître l'abstraction et favoriser la mise en place de formes diverses de communication, différents travaux enrichissent la notion de liaisons conduisant à la mise en œuvre de connecteurs [Car03, MB05], dont le comportement est explicité.

Les langages de description d'architecture (ADL) [BFCD+06, chapitre 4.2.3] sont des notations destinées à représenter l'architecture d'un système logiciel en vue de son analyse. Dans les ADLs, l'architecture d'un système est décrite principalement en terme de composants qui implémentent des interfaces, de connecteurs (interconnexions entre composants) et de leurs configurations (ou compositions). L'abstraction apportée par ces langages permet l'analyse de l'architecture (model checkers, parsers, ...), la génération de codes, la simulation de l'architecture. Nous trouvons également, associés à ces langages, l'explicitation de la sémantique des composants. Par exemple, Rapide repose sur l'ordonnancement partiel d'événements, et définit la sémantique comportementale [LV95]. Wright utilise le formalisme CSP pour analyser les connexions entre connecteurs et composants afin de détecter des deadlocks [All97].

La possibilité de créer dynamiquement de nouveaux composants et de modifier les assemblages permet l'adaptation des applications aux contextes d'usage [BF06] et en

même temps créé des besoins importants de validation des assemblages en terme de complétude (tous les composants nécessaires à l'exécution sont-ils opérationnels) [Des08], de sûreté (il ne se produit pas d'erreur inattendue), correction (assemblage sûr et conforme aux exigences non-fonctionnelles telle que la performance, la fiabilité, etc.)... [Beu05].

Les approches de vérification des assemblages à base d'algèbres de processus telles que SOFA [Pla05] ou SafArchie [Bar05] reposent sur la spécification du comportement des composants. Il est alors possible de vérifier que les composants requis ont les comportements attendus et que les composants composites respectent les protocoles annoncés [Pla05] et de contrôler l'évolution d'une architecture [BLMD06].

↪ Nous trouvons la même démarche d'enrichissement des définitions d'architectures que celle que nous trouvons au niveau des composants.
Les architectures ne se définissent plus seulement structurellement mais intègrent l'explicitation du comportement de l'assemblage. Ces données sont indispensables à la validation des assemblages et par conséquent à une intensification de l'usage d'assemblages de composants. On parle aujourd'hui de millions de composants connectés sur des grilles de calcul ; des outils prenant en charge la validation des assemblages sont indispensables à l'opérationnalisation de ces perspectives.
Nous notons également la montée en abstraction : les comportements sont spécifiés dans des langages adaptés, de plus en plus souvent indépendamment des plates-formes d'exécution.

#### 1.2.1.3 Infrastructure et indépendance plate-forme

Les assemblages logiciels reposent sur des plates-formes qui prennent en charge le cycle des vie des composants/services, les communications, les propriétés non fonctionnelles via des entités dédiées. Ces différentes entités vont constituer l'infrastructure de la plate-forme [MDBF06]. Pour pallier l'hétérogénéité des plates-formes et répondre aux besoins de portabilité et d'interopérabilité, les plates-formes sont de plus en plus souvent normalisés par des spécifications détaillées, ou même des profils UML (EJB, CCM, ...). Cependant les implémentations qui les supportent s'appuient sur des infrastructures très différentes [NBF04], rendant difficile l'intégration des services et plus généralement la compréhension de l'exécution et l'analyse des applications développées sur ces plates-formes. Lorsque les systèmes à modéliser sont critiques, des métamodèles permettent de décrire les plates-formes et ainsi diriger les générations et les validations de codes [TDG+07]. Nous retrouvons également ce besoin de correspondance entre le langage de définition d'un workflow et son exécution effective dans la thèse de M. Pourraz [Pou07] ou dans les travaux que nous avons menés sur l'intégration de services [Nan04]. Dans cette thèse, M. Nano se base sur la représentation des fonctionnalités supportées par les plates-formes à composants comme les communications, la création ou la découverte de composants, sous la forme d'une chaîne de métaobjets explicitant les différentes activités composant la fonctionnalité *(envoie, réception de la requête,...)* à la manière de Briot [Bri89] et Mc Affer [Aff95]. Ce modèle supporte un système de composition automatique des intégrations de services qui permet de détecter des conflits d'intégration et un processus de projection des intégrations de services dans différentes plates-formes à composants.

↪ La vérification de la validité des assemblages peut nécessiter de formaliser les plates-formes sur lesquels reposent ces assemblages. La description structurelle des plates-formes ne suffit alors pas. Le comportement de la plate-forme elle-même doit alors être modélisé. En travaillant à une meilleure compréhension des interrelations entre les langages et leurs plates-formes d'exécution, nous renforçons la confiance du programmeur dans l'infrastructure.

### 1.2.2 Activités et Workflow

Le terme anglais "workflow" veut littéralement dire "flux de travail". Cette forme de travail implique un nombre de personnes limité devant accomplir, en temps limité, des tâches articulées autour d'une procédure. Il y a deux grands types de workflows : le "workflow humain" et le "workflow programmatif". Le premier assure de manière transparente le suivi des tâches et leur traçabilité : qui fait quoi, quand et comment. Un non informaticien peut gérer ce type de workflow à travers un outil performant. Le second type, plus complexe, s'occupe de l'enchaînement nécessaire au cours de l'ensemble du processus. Nous nous intéressons uniquement à ce type de workflow dans ce document. Notons que dans le cadre des progiciels de gestion intégré (en anglais Enterprise Resource Planning ou ERP) les deux formes de workflows sont mélées.

La description d'un workflow caractérise les relations temporelles entre différentes activités. La description d'un workflow se fait à l'aide d'un langage dédié. Certains travaux étendent ces notions élémentaires pour intégrer : des stratégies d'itération, des contraintes de synchronisation et la gestion des erreurs.

Le workflow [3] a la responsabilité de :

- la gestion de contexte : maintien d'information entre les services.
  Le worflow prend en charge la conservation de certaines données qui sont nécessaires tout au long de la durée de vie d'un enchaînement. Par exemple, il peut s'agir de mémoriser un résultat qui est réutilisé plus tard dans le workflow. Selon le principe du couplage faible, chaque élément assemblé ignore l'existence des autres. En conséquences, les données mémorisées et les éléments du workflow forment un contexte. Il est créé au moment au lancement du workflow et se détruit lorsqu'il se termine.
- Gestion transactionnelle : commit à deux phases.
  Les éléments assemblés dans un workflow peuvent émettre des mises à jour dans les bases de données. Puisque ces éléments ne se connaissent pas entre eux, ils sont incapables de se synchroniser. C'est le workflow qui prend en charge la gestion d'une unité transactionnelle globale.
- Gestion de la logique applicative.
  Au-delà des rôles techniques de gestion de contexte et de transaction, le workflow assure un rôle applicatif. Il contient la logique fonctionnelle d'assemblage des éléments et uniquement cette logique. Les règles métiers restent localisées dans les éléments assemblés. Ainsi le workflow doit prendre en charge la gestion des échecs des activités.

Il est donc important pour la robustesse du workflow qu'il permette la définition de mécanismes de gestion des erreurs. Ces mécanismes peuvent simplement détecter l'erreur et la traiter en soumettant à nouveaux les tâches échouées ou mettre en œuvre

3. ou plutôt le gestionnaire de workflow du point de vue exécution

des actions de compensation. Ces actions de compensations permettent en cas d'erreur d'annuler ou de réparer les effets d'une tâche ayant échouée. Le traitement des erreurs permet de garder l'exécution du workflow cohérente et consistance même en présence d'erreurs.

#### 1.2.2.1 Parallélisme d'une application

Dans le cas d'une exécution d'une application sur une grille de calcul, sa description sous la forme d'un workflow en fournit une parallélisation naturelle. Trois types de parallélisme sont exploitables. Le premier et le plus direct d'entre eux est le *parallélisme de workflow*. Il correspond à l'exécution parallèle de deux services indépendants du workflow. De plus, l'exécution d'un même service sur des données différentes et indépendantes conduit au *parallélisme de données*. Ce type de parallélisme est fréquent dans les applications tirant parti des grilles de calcul, où les services doivent gérer de grandes masses de données. Enfin, deux services liés séquentiellement peuvent s'exécuter en parallèle sur plusieurs jeux de données indépendants. On parle alors de *parallélisme des services* ou de *pipelining*. Exploiter ces différents types de parallélisme permet un déploiement simple et efficace d'une application, par exemple sur une grille de calcul [GMP+06].

↪ Travailler sur le parallélisme des services fait partie de nos perspectives et n'est pas étudié dans ce document. Le parallélisme de données est acquis lorsque l'on travaille sur des services sans état, au sens où ils ne partagent pas de données de manière non explicite, ce qui est notre hypothèse dans l'application présentée au chapitre 7. L'absence d'ordre entre des activités sera toujours interprétée dans notre démarche comme du parallélisme (de workflow).

#### 1.2.2.2 Langages de description d'un workflow et Abstraction

Dans le contexte des architectures orientées services, deux points de vues sur les workflows sont possibles [Pel03] [4] :
- la spécification externe, qui décrit l'enchaînement des services et les rôles attachés à l'utilisation d'un service : c'est la *chorégraphie*,
- la réalisation interne des échanges entre services contribuant, pour le compte d'un partenaire donné, à la réalisation de la chorégraphie, que l'on appelle *orchestration*.

Une **orchestration** exprime les conditions et les enchaînements des invocations aux services. C'est le modèle des interactions que doit suivre un service pour réaliser sa fonctionnalité. Ainsi dans le cadre des architectures orientée services, ces orchestrations sont vues comme des compositions de services définissant des applications.

A l'inverse, une **chorégraphie** permet en exprimant les conditions et les enchaînements des invocations aux services de donner une vue flexible du processus global d'un ensemble de services. Les langages de description des chorégraphies tels que WS-CDL

4. The terms orchestration and choreography describe two aspects of creating business processes from composite Web services...Orchestration refers to an executable business process that can interact with both internal and external Web services. ... Orchestration always represents control from one party's perspective. This differs from choreography, which is more collaborative and allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources rather than a specific business process that a single party executes[Pel03].

sont nécessaires, tout comme BPEL, pour assurer la cohérence des comportements des points d'entrées entre les services qui opèrent.

Outre les langages dédiés à la composition de services tels que BPEL, des langages de flots ont été définis par la communauté e-Science, tels que Scufl (Simple Concept Unified Flow Language) ou MoML (Modeling Markup Language). Ces langages sont conçus pour la description du flux de données. Des opérateurs de composition de données d'entrée tels que *dot* et *cross product* permettent en particulier une description précise de la manière dont un service est itéré sur ses entrées. A la différence des langages issus de la communauté du e-Business tels que BPEL, ce type de langages n'intègre que quelques opérateurs de contrôle sommaires, la logique de l'application étant décrite à l'intérieur des services eux-mêmes.

**Formalisation et validation** Différentes techniques formelles peuvent être utilisées pour modéliser et valider des workflow : algèbres de processus [ABV04], LOTOS (CADP) ou automates [EBR00].

Afin de permettre la validation des assemblages de services, différents travaux tendent à dériver du langage BPEL4WS des vérifications formelles. Parmi ces travaux, dans [CCCV05, Mar05] les auteurs proposent une transformation du langage WSBPEL vers la notation CSS pour l'un et les réseaux de Petri pour l'autre. Les auteurs utilisent cette formalisation pour vérifier la "compatibilité des services" à assembler, ou la propriété d'interchangeabilité des services afin de remplacer des services en prenant en compte à la fois les aspects "syntaxiques" et le comportement des services. Notons que ce travail exige une formalisation de l'ensemble des services mis en jeu dans une orchestration [Vir04, FUMK04].

Dans [Pou07], la description de l'architecture est totalement formelle et son exécution repose sur le pi-calcul. Ce langage, pi-diapason, a un pouvoir d'expression plus grand que BPELWS. Sur cette base, le langage diapason-* permet de formuler des propriétés spécifiques à une orchestration écrite en pi-diapason et de vérifier ces propriétés, même en fonction des évolutions dynamiques des orchestrations. Parmi les propriétés nous pouvons distinguer les propriétés de sûreté (sous certaines conditions quelque chose de mauvais n'arrivera jamais) et les propriétés de vivacité (sous certaines conditions, quelque chose de "bon" finira par arriver). Par exemple de propriétés, l'unicité d'une invocation (une et une seule facture est produite), il n'y a pas d'action de facturation qui ne soit suivie d'une livraison et une livraison n'est pas possible si elle n'a pas été précédée d'une facturation.

**Moteur de workflow** L'exécution de la description d'un workflow sur un jeu de données d'entrée est assurée par un moteur de workflow, responsable en particulier de la transmission des données entre les différents services.
De nombreux moteurs de workflow sont utilisés pour le déploiement d'expériences scientifiques "in-sillico" dans des domaines applicatifs variés dont Taverna [OAF+04] et MOTEUR [GMP+06] qui est développé dans l'équipe Rainbow. Il se concentre sur l'exploitation des différents types de parallélismes pour le déploiement d'applications sur une grille de calcul. Dans le monde des architectures SOA, différents moteurs de workflows exécutent des orchestrations BPEL. Nous avons utilisé dans nos développements APACHE ODE.

↪ Les langages de description de workflows sont pour l'essentiel indépendants des plates-formes et donc des moteurs ; des opérateurs spécifiques sont définis comme du sucre syntaxique pour simplifier les expressions de workflows et favoriser certaines optimisations des moteurs. C'est sur la base de ces langages que sont construites, éventuellement par transformation vers d'autres formalismes, les vérifications de workflows.

#### 1.2.2.3 Diagrammes de comportements en UML

Les diagrammes UML 2.0 [OMG07] sont classés en diagrammes de structure et de comportements. Dans notre contexte d'étude, seule la seconde catégorie nous intéresse. Les diagrammes de cas d'utilisation capturent les exigences fonctionnelles et sont réalisés par les diagrammes de séquences. Les diagrammes de machines à états explicitent les états possibles d'un objet et les transitions entre ces états. Un diagramme d'états montrant en général le comportement d'un seul processus, il est alors très difficile de comprendre l'enchaînement entre processus parallèles. Les diagrammes d'états sont peu appropriés à expliciter les orchestrations, et surtout les chorégraphies. A l'inverse les diagrammes d'activités sont utilisés pour décrire les workflow ou la logique procédurale. Les diagrammes d'interactions permettent eux de modéliser les interactions possibles entre différents objets. Les interactions mettent en jeu à la fois le séquencement entre les messages et chemins de communications. Des diagrammes de temps (Timing diagrams) visualisent les contraintes de temps entre les envois de messages.

Les ***diagrammes d'activités*** permettent de spécifier un ensemble d'activités et leurs enchaînements. La notion d'activités composites est particulièrement intéressante, elle permet de donner d'une activité une vue boîte noire (dans une activité englobante) et une vue boîte blanche détaillant le workflow interne. Différents travaux ont donné une sémantique formelle aux diagrammes d'activités, sémantique en général basée sur des variantes des réseaux de Petri ou des systèmes de transition [EW04].
Les ***diagrammes d'interactions*** capturent également bien le comportement sur l'ensemble d'un système. La sémantique d'une interaction est donnée par une paire d'ensembles de traces représentant les traces valides et invalides respectivement. Une trace dans le cadre des diagrammes d'interaction correspond à une séquence d'occurrences d'événements.

Dans [VDS05, page 73], l'auteur utilise ces traces pour identifier les inconsistences d'interactions lors de la spécialisation des interactions. En particulier il y a inconsistence d'interaction d'invocation lorsque que l'ensemble des traces reçues au niveau d'un parent n'est pas un sous-ensemble des traces reçues au niveau d'un fils.

↪ UML 2.0 intègre la notion d'activité. Celle-ci généralise le concept d'activité élémentaire et d'activité de contrôle de flots. Pour des raisons de composition nous nous en démarquons au chapitre 2. La modélisation des traces valides nous permet en effet de capturer plus aisément des algorithmes complexes de composition.

### 1.2.3 Séparation des préoccupations

Jusqu'à présent nous avons essentiellement abordé la composition comme extérieure aux entités composées.

Nous proposons à présent de nous intéresser aux compositions qui modifient les entités mises en jeu. En effet, la complexité croissante des applications et l'expertise nécessaire dans les différents aspects techniques ont conduit à une nouvelle approche du développement de logiciels basée sur la "séparation des préoccupations" tandis que des outils dits "Tisseurs" (Weavers en anglais) se chargent de valider la cohérence du tout et leur composition. La programmation par séparation des préoccupations connaît un engouement certain. En effet, elle apporte un niveau d'abstraction supplémentaire, qui favorise à la fois la robustesse du développement, la rapidité de production et la réutilisation des codes [KM05]. Ces critères sont vrais pour autant que l'on associe à ces techniques des outils de validation et que l'utilisateur ait confiance dans leur production, i.e. les codes générés et les compositions des codes.

Les conséquences de la séparation dans la composition s'expriment différemment selon le point de vue que l'on a sur le système à construire.
Lorsque l'on se positionne dans une séparation dans le temps, la séparation doit permettre un développement incrémental en ayant connaissance des développements ultérieurs. Les activités sont alors modifiées par incréments. Nous sommes dans une démarche "feature" [BSR03, LHBL06].

Si au contraire la séparation est induite par l'intervention de plusieurs développeurs de manière concurrente sur un même ensemble d'activités, la difficulté est de savoir composer ces différents changements concurrents sans l'expression d'une connaissance les uns des autres. Cette forme de séparation est nécessaire à l'intervention de plusieurs experts sur une même application. Chacun a sa vision de l'application, et introduit des éléments propres à son expertise (sécurité, persistance, ...). Ces changements peuvent intervenir à l'exécution pour adapter une application. Avec la thèse de Stéphane Ducasse [Duc97] dont j'étais co-directeur, nous abordions déjà ce point d'un point de vue "interactions" au niveau du langage Clos[DBFP95].

Si nous considérons ces changements comme l'ajout d'activités qui complètent les activités existantes, nous sommes dans une démarche aspects. Elle s'accompagne souvent d'une généralisation du processus ; le développeur ne déclare pas précisément les entités à modifier, il les définit par leurs caractéristiques (nom, arité, temps d'exécution, ...).

Enfin lorsque le système à construire se définit sans modification d'un assemblage existant, mais par réutilisation d'applications existantes, la composition est alors associée à de l'intégration.

On peut distinguer deux approches du développement par séparation des préoccupations : le développement par *sujets* et par *aspects*.
La notion de sujet est proche de la notion de vue et introduit un découpage vertical. Les mécanismes de composition s'appuient en particulier sur la fusion [BSR03].
Les aspects introduisent un découpage transversal et la composition repose sur des points d'entrelacements des codes. Parmi les aspects souvent traités nous trouvons : la gestion des utilisateurs (authentification), l'archivage des données (persistance), la programmation concurrentielle (multi-threading), l'information pendant l'exécution du logiciel (trace), l'application de patterns de programmation, etc.

#### 1.2.3.1 Programmation par Aspects et Interactions

Il existe différentes mises en œuvre de la programmation par aspects [BSL01] : approche par transformation de programme (AspectJ [KM05], programmation par attributs [RPPM06]), approche par transformation d'interprète ou des moteurs d'exécution des workflows [RBY+04], approches hybrides (Javassist[5], etc.). Nous renvoyons le lecteur intéressé par ce sujet au rapport FAROS 1.1 [BFCD+06] pour plus de détails généraux et principes de la programmation par aspects.

Nous noterons cependant que la programmation par aspects s'applique aujourd'hui à de nombreuses formes de programmation au niveau de la programmation même des éléments de l'assemblage [PSDC06], des assemblages qu'il s'agisse d'assemblages de composants [BFEO+02] ou de services [CM05, CM04, CF05b], des infrastructures [CF05a, VVJ06] ou des diagrammes de séquences [KFJ07].

Les aspects jouent un rôle tellement important que peu à peu les greffons (advices) font intervenir de nouvelles entités qui encapsulent la logique du greffon (par exemple des services de gestion de la sécurité ou de la QoS). Cette approche permet également de limiter le langage de définition des aspects à des instructions de coordination [CM04].

Un des objectifs de la programmation par aspects est de rendre transparent aux utilisateurs l'intégration de contrôles. Or la composition des aspects intervenant sur un même point d'interception ne peut pas toujours être gérée sans intervention de l'utilisateur, de plus le choix d'un point d'interception pour un aspect peut empêcher l'application d'un autre aspect [KFJ07]. Cette tâche est alors d'autant plus compliquée que tous les aspects n'ont pas nécessairement été définis par lui et que les conflits peuvent apparaître alors qu'il définit simplement de nouvelles classes ou de nouveaux aspects [TBB04]. La prise en compte de l'ordre des adaptations va alors à l'encontre de la séparation puisque par définition les acteurs d'une adaptation ne se connaissent pas nécessairement. Il apparaît donc nécessaire de définir les adaptations selon des algèbres qui gèrent la composition indépendamment de l'ordre des déclarations et détecte les conflits éventuels [LHB05]. Les travaux relatifs aux interactions entre aspects s'intéressent tout particulièrement à ce dernier point [FF05, PSDB04, DFS04, DFL+05, LHB05, KFJ07].

Finalement, nous retrouvons le problème de l'analyse du code généré lors des phases de mise au point des logiciels (débogage, test). Les outils autour des langages tels que ceux définis autour de AJDT, basé sur AspectJ, permettent néanmoins de passer de façon transparente, en mode débogage, du code d'une classe à celui d'un aspect.

↪ La programmation par aspects permet une formalisation des contrôles séparément des codes à contrôler. Ces contrôles doivent être composés lorsqu'ils interviennent sur un même point d'exécution. Dans ce cas, différentes solutions sont proposées : ordonnancement par l'ordre de déclaration ou par priorité d'un aspect sur un autre, détection de conflits, composition semi-automatique, ... De manière générale et malgré une recherche de transparence, il reste nécessaire de contrôler la composition des aspects à la fois pour calculer le résultat des compositions et pour le valider.

5. http ://www.csg.is.titech.ac.jp/ chiba/javassist/

### 1.2.4 Compositions

Nous pouvons distinguer deux formes de compositions qui sont fréquemment discutées : la composition par surimposition et la composition par quantification [ALB+07]. Dans le premier cas, les éléments à composer sont déterminés par "pattern matching", correspondant à la reconnaissance d'éléments communs par exemple en utilisant le nom, le type, la position dans un arbre, plus généralement les signatures[FFR+07]. Dans le second cas, les éléments à composer sont identifiés par quantification, c'est à dire que les éléments à composer sont cités, par exemple par des points de coupes. La nuance est, de notre point de vue, essentiellement dans la démarche. En effet, la première forme de composition est le plus souvent utilisée en identifiant les éléments à composer par égalité. Dans ce cas, les éléments sont composés deux à deux. Dès lors que l'on introduit l'équivalence, plusieurs éléments peuvent être sujet à la composition, comme dans le cas de la quantification. La différence vient de ce que dans le premier cas les éléments jouent tous le même rôle sans connaissance a priori de leur composition potentielle, alors que dans le deuxième cas, certains éléments jouent le rôle de modificateurs et explicitent les points d'accroches.

Nous nous intéressons essentiellement à la composition d'activités par surimposition, mais lorsque la composition par quantification est nécessaire (par exemple dans la composition d'interactions entre composants cf. §6), nous explicitons le rapprochement à l'instar des travaux sur le développement orienté "Feature" [ALB+07].

#### 1.2.4.1 Composition de workflows

De la même manière que se développent des lignes de produits, des workflows sont définis comme des services réutilisables et standardisés [PS05]. Les workflows doivent alors être modifiés et composés pour répondre à des besoins spécifiques.

La composition de workflows peut alors être appréhendée sous deux points de vue complémentaires :

- Le premier en considérant les éléments du système (services et orchestration) comme des boîtes noires accessibles uniquement via leur interface. Dans ce cas, la définition du résultat d'une orchestration comme nouveau service permet une composition récursive des orchestrations [KMW03]. Cependant cette composition pose des problèmes d'appels multiples aux services communs, d'absence de partage du contexte de gestion des erreurs délocalisées, etc. Nous avons étudié ce point pendant le DEA de Clémentine Nemo [Nem06], puis au travers d'une application de workflow dans le cadre d'une application de traitement d'images sur grille [NGBFM07].
- Le second en considérant les orchestrations comme des boîtes blanches qu'il s'agit de composer par tissage de codes. Dans ce cas, la composition d'orchestration est une tâche plus complexe, qui suppose une bonne connaissance des services et orchestrations existantes. Ce dernier point fait l'objet de la thèse de Sébastien Mosser sous ma co-direction. Par une approche nouvelle basée sur la définition d'évolutions, il propose de modifier des orchestrations [MBFR08] et de contrôler leurs évolutions par composition.

**Validation de composition de workflows** La notion de compatibilité entre les orchestrations afin de permettre leur composition est une notion définie par [Mar05] en

terme de workflow. Cette notion est découpée en deux sous notions : la compatibilité syntaxique et la compatibilité sémantique. La première est une notion assez simple : deux workflows sont déclarés syntaxiquement compatibles si les deux processus sont disjoints (ils ne font pas appel aux mêmes services) et pour chaque point d'entrée/sortie en commun, l'entrée de l'un correspond à une sortie de l'autre et vice versa. La compatibilité sémantique selon l'auteur[6] correspond à analyser le flot d'activités défini par chaque orchestration et à détecter les inconsistances tels que des interblocages.

Dans [PS05], les workflows sont définis selon le formalisme des réseau de Petri. Un réseau de Petri se définit par des places, des transitions entre ces places et une relation de flot. Afin de supporter la composition des workflows les opérations de sélection de sous-réseaux , de regroupement, de différences et de jointure sont définies. La jointure se définit par une jointure naturelle lorsqu'elle consiste à fusionner des places ou des transitions et de jointure "théta" lorsqu'il s'agit d'ajouter des connexions entre des places et des transitions de deux réseaux disjoints. Cet ensemble d'opérateurs forme une algèbre sur laquelle sont prouvées : la commutativité de l'union, la possibilité de définir l'union naturelle par sélection et jointure théta, ... Cette approche permet de définir une forme normale du workflow, sur laquelle peuvent être détectées des anomalies telles que : la redondance (une information est répétée inutilement dans le workflow), les anomalies de mises à jour (par exemple, un ordre entre deux places est modifié ; ce nouvel ordre doit être respecté dans tous les workflows qui y font référence), un comportement inconsistant. Les dépendances du flot sont alors analysées pour détecter ces anomalies.

$\hookrightarrow$ Les travaux sur la composition de workflow mettent en avant la nécessité d'une validation des résultats des assemblages et la nécessaire compréhension des différents éléments qui composent un workflow pour aboutir à cette validation. La composition de workflows est complexe et relève d'opérations sur des graphes. Ces deux points s'opposent à la vision "boîte noire" des orchestrations comme de nouveaux services. Dans l'application décrite au chapitre 7 nous proposons une aide à la composition de workflow via la reconnaissance d'activités redondantes entre autre et l'utilisation de composition de transformations à l'instar d'opérateurs sur des workflows.

#### 1.2.4.2 Composition de modèles

Avec l'évolution du développement dirigé par les modèles, un même logiciel est décrit par plusieurs modèles liés. Ces modèles peuvent représenter le même sous-système ou des sous-systèmes différents créés en parallèle par plusieurs concepteurs ou une combinaison des deux. Il est alors nécessaire de mettre en place des outils pour comparer et fusionner différents modèles en un nouveau. Ainsi la composition de modèles peut faire intervenir aussi bien de la composition d'assemblages à l'instar des travaux évoqués au paragraphe 1.2.1 que de la séparation des préoccupations.

**Ordonnancement des transformations et calcul de différence** Dans [AP07], les auteurs s'intéressent aux développements concurrents autour d'un modèle original initial. Ils proposent de définir la composition de modèles sur la base d'un calcul de *différence* (noté -) et une opération d'*union* (notée +). A partir d'un modèle original $M_{original}$ et de modèles dérivés $M_1$, $M_2$ par exemple, la composition vise à construire un modèle

6. peut-être mieux nommée compatibilité comportementale

final $M_{final}$ tel que :
$M_{final} = M_{original} + ((M_1 - M_{original}) + (M_2 - M_{original})) = M_{original} + (\Delta_1 + \Delta_2)$.

Les éléments concernés par le calcul des différences sont identifiés par l'égalité des noms. Une différence (notée $\Delta$ dans notre exemple) s'exprime alors sous la forme d'un ensemble de transformations atomiques. L'union consiste à modifier le modèle par application des transformations. L'application des transformations est ordonnée selon un ordre pré-calculé. Lorsque plusieurs unions s'appliquent, il convient donc de les composer afin d'assurer l'associativité des compositions, ce qui conduit à la définition d'un opérateur de composition des différences :$\otimes(\Delta_1, \Delta_2) = \Delta_1'$

D'autres travaux relatifs aux compositions de transformations tels que [MKR06] apportent également des solutions à la détection et l'ordonnancement des transformations pour les rendre indépendantes de l'ordre d'application.

**Fusion en UML et dépendance domaine** La fusion définie par UML porte sur la fusion de classes, packages, associations, propriétés, etc. La fusion de graphes d'activités n'est pas proposée parce que complexe et dépendante domaine [OMG07, page 116]. De manière générale, la fusion n'est possible que si des préconditions sont vérifiées sur les éléments sujets à la fusion afin d'assurer que les capacités des éléments résultant de la fusion ne sont pas réduites [7]. La fusion opère par identification des éléments à partir de leur nom. Dans [AD06], les auteurs discutent ces restrictions et proposent d'étendre la vérification des préconditions à la vérification de contraintes définies par l'utilisateur et la reconnaissance des éléments à fusionner sur la base de correspondances données par l'utilisateur.

**Détection semi-automatique des correspondances** Dans [Pas06b], les auteurs définissent la composition de deux modèles sur la base d'un modèle de *correspondance* calculé par une opération *Match* qui prend en paramètre les modèles à composer et retourne le modèle de correspondance. La composition des modèles en elle-même repose alors sur la "fusion" des paires d'éléments qui matchent (cf. page 15 et figure 2 de [Pas06b]), la transformation des éléments qui ne matchent pas et une opération de composition.

Dans Atlas Model Weaver [8] [FV07] est posé le problème de la détermination des correspondances entre les éléments des modèles à composer. Sur la base d'un grand nombre d'expériences, l'intervention éventuelle d'un humain pour décider des compositions à appliquer est soulignée. L'utilisation d'outils extérieurs pour décider des similarités est également proposée, par exemple pour calculer la distance entre des modèles. Ce point est particulièrement utile si trop de similarités sont détectées et qu'il faut choisir parmi celles-ci les compositions à opérer.

Dans [NSC+07], les auteurs s'intéressent à la détermination de correspondances entre les états de machines à états. Ces correspondances sont établies sur la base d'heuristiques à la fois sur le nom des états (correspondance typographique et linguistique) et également sur le comportement associé aux états via l'étude des transitions. Ce travail permet d'établir une base de correspondances que l'utilisateur doit cependant confirmer et compléter.

7. a resulting element will not be any less capable than it was prior to the merge.
8. Le terme Weaver est utilisé ici dans un sens différent de celui introduit précédemment. Le tissage consiste ici à déterminer les liens entre des modèles.

**Conflits de composition** La détection des conflits de composition est classifiée dans [Pas06a] selon les conflits du niveau modèle qui sont alors domaine dépendant, des conflits de métamodèles qui peuvent interdire la composition, des conflits de méta-méta-modèles qui peuvent engendrer des compositions non conformes au métamodèle cible.

**Composition des modèles et réutilisation de codes** Dans [BGB05], le problème de la composition de modèles indépendant plates-formes (PIM) et la réutilisation des modèles spécifique plates-formes précédemment définis (PSM) et les codes correspondants est très clairement posé. Les auteurs proposent que les modèles composés ne soient pas modifiés directement par la composition. L'expression des éléments additionnels de composition via des règles de composition sert alors de base à l'expression de la composition au niveau du modèle et à la génération de "glue" entre les PSMs préexistants. Les correspondances dirigent alors les compositions qui reposent sur un pré-ordonnancement des règles de composition et des structures de contrôles.

**Correspondances et compositions** S. Clarke [Cla02] étend UML afin d'y intégrer une relation de composition. Plus précisément, cette extension modifie la hiérarchie des classes du métamodèle d'UML pour introduire plusieurs catégories d'élément composable (*ComposableElement*) ; les propriétés, les associations, les opérations, les classifieurs et les sujets (spécialisation d'un paquetage) en font notamment partie. La hiérarchie des relations offertes dans UML est, elle aussi, modifiée afin d'intégrer plusieurs catégories de relations de composition à partir desquelles il est possible d'établir une correspondance entre les entités à composer. Naturellement comme pour tout élément du métamodèle UML, la description de leur sémantique est complétée par la définition de contraintes et de règles de validité. Une fois la correspondance établie entre deux éléments composables (ils peuvent avoir ou non les mêmes noms), il est possible d'associer à la composition le comportement d'une fusion ou d'un remplacement. Lorsque les éléments composables sont constitués d'autres éléments (c'est le cas des classifieurs) et si la correspondance entre certains de ces éléments n'est pas explicite alors c'est le nom qui détermine si ces éléments sont candidats à être remplacés ou fusionnés. Lorsque les noms diffèrent, ils sont ajoutés sans changement au modèle résultat. Concernant la fusion il est proposé plusieurs solutions pour résoudre les conflits (précédence, transformation, redéfinition, etc.). Cependant la fusion des assertions d'une opération ou d'une classe n'est pas vraiment abordée et surtout la composition basée sur une description graphique peut se révéler complexe et n'est pas réutilisable.

Dans [FFR+07], les auteurs proposent une approche de la composition des modèles qui étend les travaux précédents sur la détermination des correspondances. L'algorithme de composition consiste à utiliser des fonctions de comparaison des signatures des éléments des modèles. Les éléments des deux modèles à composer sont comparés et dans le cas où leur signatures sont considérées comme équivalentes, ils sont fusionnés toujours en appliquant des fonctions spécialisées. Ainsi nous trouvons dans cette approche à la fois les avantages d'une approche automatisée de la composition et en même temps la flexibilité nécessaire à la composition de modèles issus de différents domaines. Notons que de plus afin de faciliter la reconnaissance des éléments communs, une phase dite de pre-merge directives permet à la personne qui compose les modèles d'adapter les modèles, par exemple par renommage des éléments afin qu'ils soient considérés comme équivalents. De même une phase dite de post-merge directives permet d'adapter le mo-

dèle résultant de la composition par exemple pour retirer des associations qui auraient dues disparaitre.

↪

- la composition des modèles repose sur la détermination de correspondances ; celle-ci dépend du domaine ciblé et ne se base pas seulement sur des correspondances structurelles ;
- l'intervention humaine dans le choix des compositions proposées ou la détermination des correspondances est nécessaire dans certains domaines ;
- il est important d'identifier les conflits de composition à la fois pour assurer la cohérence des compositions et pour anticiper l'élaboration de règles de composition qui pallieront ces situations ;
- la composition des transformations relève d'un procédé qui peut être complexe et retardé ;
- l'opération de transformation telle qu'énoncée dans ces travaux suppose que les éléments non intervenant dans une opération de matching ne sont pas impactés par les fusions. Dans le cadre de la composition d'activités, cette hypothèse n'est pas valide. De même, les problèmes dus aux éléments partagés par plusieurs correspondances et les impacts sur les éléments simplement reproduits ne sont pas explicités ; ces points constituent cependant une des difficultés de la composition des activités ;
- au niveau des modèles, la conception par séparation des préoccupations favorise la collaboration et l'enrichissement du modèle de base, tout en opérant des contrôles non supportés par les ateliers standard de modélisation qui reposent sur la seule sémantique d'UML. En enrichissant ces capacités de validation, la modélisation par séparation des préoccupations laisse envisager une intensification de cette forme de développement. Elle allie déclarativité et génération de code, de la sorte que l'expressivité et la robustesse des applications y gagnent. Néanmoins cette puissance potentielle ne sera réalité que si les outils développés assurent des propriétés de composition "intuitive", démontrables et probablement non dépendantes d'une relation d'ordre des déclarations. La détection des conflits de composition est alors essentielle.

### 1.2.5 Langages et métacompilation

Les travaux sur la compilation reposent sur une analyse des données conduisant à une représentation inductive intermédiaire qui sert de base aux vérifications et aux générations de codes.

Le choix de la représentation intermédiaire correspond en compilation à un " relèvement " ou structuration d'un programme linéaire. Le code source est le point départ. L'objectif est de le transformer en du " code ". La représentation intermédiaire correspond généralement à un arbre de syntaxe abstraite. Pour effectuer les calculs, cet arbre est annoté par des attributs et/ou des structures additionnelles comme une table des symboles. Le choix de ces annotations et leur valuation constitue une étape essentielle de la construction du compilateur. Ces éléments additionnels et leur valuation capturent une part de la sémantique du langage. Par abus de langage, nous qualifions cette représentation de sémantique. La partie arrière du compilateur est alors dépendante à la fois de cette représentation au niveau de laquelle les calculs peuvent être réalisés et de la

cible pour laquelle les codes sont générés. Cette dépendance est capturée dans les grammaires attribuées par l'annotation de la grammaire par des attributs et la définition d'une " interprétation " qui informellement se définit par un système de type et de fonctions définies sur ces types. De fait il est alors possible d'aborder la compilation à la fois du point de vue du concepteur du compilateur qui dispose des accroches pour énoncer les mécanismes de transformations et du théoricien pour vérifier les propriétés générales de ces mécanismes.

Nous avons explicité le rapprochement entre le domaine des langages et la métamodélisation dans [BFFZ06], et plus particulièrement décrit la nécessité d'un raisonnement au niveau des modèles dirigés par les impératifs des domaines dans [BFFN05].

↪ Le domaine des langages a défini des théories et des outils pour capturer les éléments fondateurs permettant de construire une représentation intermédiaire sur la base de laquelle les opérations de " compilation " peuvent opérer. Les grammaires attribuées ont ouvert la voie à la **métacompilation** [FZ82] en considérant que l'évaluation pouvait être dirigée par des informations additionnelles, les attributs, que l'ordre des évaluations pouvait être pré-calculé indépendamment du domaine cible et que les évaluations dépendantes domaines pouvaient être capturées par des fonctions définies sur le domaine d'interprétation. Le choix de la représentation sémantique n'est pas automatique de manière générale et dépend évidemment du domaine initial et des objectifs visés. Dans [Kli08], Paul Klint explicite la complexité de ce choix.

### 1.2.6 En résumé

La composition est un élément essentiel à la réalisation des applications futures, mais elle requiert un support important afin de répondre aux besoins de validation des assemblages (passage à l'échelle, propriétés de sûreté, vivacité, etc.), de programmation proche des spécificités des programmeurs (langages d'architectures, orchestrations, algèbre de processus, langages d'aspects, etc.) et de flexibilité pour assurer une programmation indépendante des supports et adaptable en fonction des particularités du contexte d'exécution.

## 1.3 Démarche

Les travaux présentés dans ce document abordent différents domaines du génie logiciel et utilisent des techniques différentes pour être en adéquation avec les problèmes abordés. Il sous-tend cependant à ces études un même principe qui repose sur l'abstraction des systèmes étudiés en terme d'activités, des mécanismes de composition qui supportent la séparation dans l'expression des activités et la gestion des conflits. Les activités sont représentées différemment en fonction des applications : algèbre de processus, langage BPEL, notations graphiques. Les algorithmes de composition, quand ils existent, dépendent de ces représentations.

Si nous prenons le point de vue du développeur de ces environnements de composition, qu'il s'agisse de composer des aspects, des interactions ou des workflows, il est amené à extraire des langages les éléments porteur des informations à composer, définir voire adjoindre à ces éléments des informations additionnelles pour diriger la composition, définir les algorithmes correspondant, établir les propriétés qui doivent être véri-

fiées puis vérifier qu'elles le sont effectivement dont la clôture, l'indépendance de l'ordre des compositions, l'associativité, ....

La tâche est d'autant plus difficile que certaines compositions, par exemple les aspects, ne se résument pas à un "copy&paste". Des éléments disparaissent (e.g. consécutivement à un aspect ne contenant pas de *proceed*), d'autres sont ajoutés ; la composition peut alors introduire des problèmes tels que de la redondance ou les inter-blocages.

La représentation inductive des assemblages d'activités est alors nécessaire à la fois pour exprimer les opérations de modification et construction des assemblages et pour raisonner sur ces assemblages.

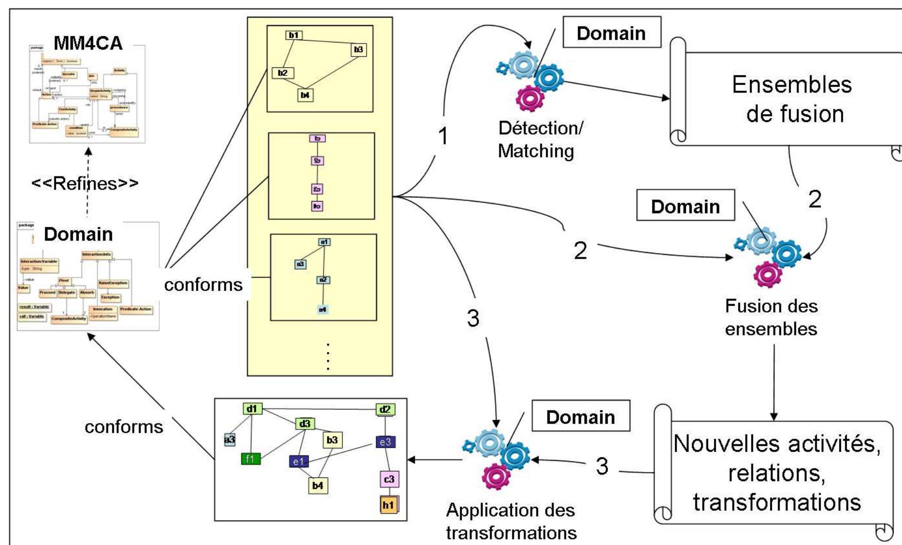



FIGURE 1.2: COMPOSITION D'ACTIVITÉS : POINT DE VUE GÉNÉRAL

Les langages définissant les "activités" constituent donc à la fois mon domaine de départ et mon domaine cible. A l'instar des travaux sur la compilation, la première étape a donc été, par "abstraction" sur la base de plusieurs travaux sur la composition d'activités, de déterminer une représentation intermédiaire qui capture à la fois la structure inductive des activités composites (analogie avec l'arbre de syntaxe abstraite) et localise les informations additionnelles nécessaires à la prise en compte de la sémantique des flots d'activités ciblés (analogie avec l'introduction des attributs)(cf.1.3.1). La représentation intermédiaire est un graphe étiqueté dirigé acyclique. Les étiquettes sur les sommets capturent les informations additionnelles. Des contraintes sur ce graphe précisent les contraintes propres au domaine.
La seconde étape a été la définition de l'évaluation de cette représentation pour établir la composition des activités. Cette évaluation utilise différentes fonctions dépendantes du domaine et qui opèrent sur cette représentation intermédiaire "étendue" en fonction du domaine : ces fonctions définissent l'interprétation sur domaine. Ces fonctions sont contraintes et doivent respecter des propriétés assurant la terminaison de l'algorithme de composition, l'absence de circuit et la clôture de la composition. Cette étape est à rapprocher des travaux sur la transformation de modèles (cf.1.3.2). D'autres propriétés peuvent être exigées pour assurer par exemple la commutativité des compositions ou la préservation des activités, . . . (cf. 1.3.3).

### 1.3.1 Abstraction et Interprétation

A l'instar de l'interprétation abstraite d'un langage de programmation ou de spécification, nous nous intéressons à établir les sémantiques des compositions liées par des relations d'abstraction. La sémantique la plus précise décrit l'exécution réelle de la composition de manière très fidèle, elle est la sémantique concrète. Pour la détermination, de certaines propriétés nous n'avons pas besoin d'une telle précision. Nous utilisons l'abstraction des modèles pour restreindre la vérification aux éléments essentiels. Lorsque l'abstraction nous fait perdre de la précision, nous utilisons des "fonctions sur domaine" qui permettent de calculer les éléments nécessaires au raisonnement global au niveau du domaine. L'ensemble de ces fonctions constitue l'interprétation sur domaine par analogie avec les grammaires attribuées [FZ82]. Cette approche permet d'aborder les calculs de complexité, de donner une approche unifiée de la composition et d'établir un schéma général pour la vérification des propriétés.

### 1.3.2 Compositions d'activités : un point de vue transformation de modèles

#### Modèle : consistance, conformance et équivalence

Un modèle est *consistant* s'il existe au moins un système qui respecte l'ensemble des propriétés définies par le modèle [HKR+07].

A chaque modèle présenté dans ce document correspondent des applications (système). Tous les modèles sont donc consistants. Les modèles sont vérifiés et transformés dans des systèmes tels que des interactions entre composants ou des orchestrations de web services. Un modèle est toujours défini dans notre travail relativement à un métamodèle de référence, et nous notons cette *conformance* par l'appartenance : $m \in MM$ signifie que le modèle $m$ est conforme au métamodèle $MM$ [9] [BFFZ06]. C'est au niveau du métamodèle que nous définissons les vérifications et transformations qui seront appliquées aux modèles qui lui sont conformes. Nous définissons donc un métamodèle d'activité (cf. §2) que nous raffinons pour l'adapter à différents domaines cibles (cf. partie II).

MOF 2.0 définit principalement trois mécanismes d'extension des métamodeles : la spécialisation de classe, l'union et les sous-ensembles de propriété, et la redéfinition de propriétés. Un des objectifs est que les fonctionnalités (entre autre transformations) qui ont été définies sur un métamodèle M restent utilisables sur des modèles dont le métamodèle est une extension M' du métamodèle M. La sémantique des extensions est encore obscure et ne permet pas de considérer le sous-typage des modèles comme nous le ferions par polymorphisme dans le domaine des objets [SJ05, AP07].

Dans ce document, nous nous limiterons à une lecture "objet" des relations entre un modèle et un métamodèle et préciserons l'appartenance d'un modèle d'activité à un domaine à la fois par sa conformité structurelle vis-à-vis de son métamodèle de référence et une fonction de normalisation.

L'*équivalence* des modèles est nécessaire à établir des comparaisons entre les résultats issus des compositions. Certaines équivalences ne peuvent être établies qu'au niveau des métamodèles spécialisés dans les domaines. En effet, les métamodèles induisent des représentations qui peuvent syntaxiquement différées sans différer sémantiquement conduisant à des réalisations équivalentes [HKR+07].

9. Les concepts de modèles et métamodèles sont considérés connus des lecteurs. Pour plus d'informations sur ces notions nous renvoyons le lecteur à [JFBF06].

#### Opération de composition de modèles

Dans sa forme générale, la composition se définit comme une opération qui prend en entrée des modèles et renvoie un nouveau modèle ou échoue. Soient $MMs$ et $MMc$ deux métamodèles, une opération de composition notée $\bigoplus$ est définie de $MMs^{+} \longrightarrow MMc \times \Upsilon$. Elle prend en entrée des modèles conformes au métamodèle source $MMs$ et renvoie un modèle conforme au métamodèle cible $MMc$ ou échoue.

Nous avons réduit notre recherche en considérant uniquement des modèles conformes à un même métamodèle. Ainsi sur la base d'une conformité des modèles initiaux, l'opération de composition doit assurer la conformité du modèle résultant de la composition.

La construction de ce modèle résultat de la composition procède par transformations des modèles initiaux en de nouveaux modèles.

Contrairement aux approches où la composition des modèles trouve son correspondant dans les systèmes qui réalisent les modèles, nous ne cherchons pas une telle propriété. En effet, la complexité des activités et de leur réalisation est une motivation à leur modélisation. L'expression de leur composition au niveau des modèles est notre objectif, nous ne prétendons pas que cette composition existe au niveau des systèmes, en cela nous nous démarquons de la définition générale donnée par Herrmann et co. [HKR+07]. Nous basons ce travail sur l'existence de transformations entre les métamodèles qui font l'objet de notre étude et les plates-formes cibles, pas sur la transformation des compositions elles-même.

**Transformations :** L'opération de composition en elle-même requiert différentes transformations en fonction des domaines cibles. Ainsi elle peut être réalisée à la main. Dans ce cas, elle est difficilement reproductible ; la logique de la composition n'est pas exprimée. Les outils peuvent néanmoins vérifier a posteriori différentes propriétés dont la conformance du modèle résultat, la préservation des propriétés, la mémorisation des actions utilisateurs [dF07]. A l'inverse la composition peut être totalement automatisée, c'est ce que propose la programmation par aspects qui compose un aspect dans du code et génère automatiquement un nouveau code. L'aspect contient les informations nécessaires à la composition. Nous trouvons dans [Wag08] une approche par surimposition pour composer des transformations. Ces compositions automatiques reposent sur des opérateurs tels que *before*, *around*, ... ou dans le cas des modèles *merge, override*, ou *extends*. De manière intermédiaire, différents travaux proposent d'interagir avec le programmeur pour décider des compositions[FV07]. Ces travaux permettent de mettre en relief les étapes suivantes à la composition : identification des éléments de modèles à composer (expression des correspondances), détermination des transformations à appliquer et application des transformations. Nous nous situons dans cette démarche. En fonction des domaines, ces étapes seront automatiques ou non.

Les décisions que nous prenons relativement aux transformations et aux contraintes sur domaine impactent les propriétés de la composition (cf. 1.3.3). Nous nous attachons dans ce travail à caractériser les propriétés que l'on peut attendre des compositions et leur impact sur la modélisation du domaine et les choix de transformations.

Le respect de certaines propriétés lors de la composition d'activités peut impliquer de détecter des *conflits* et d'interrompre la composition.

La nécessité de raisonner sur les activités et le besoin d'inférence nous ont conduit à définir les transformations sous la forme de programmes Prolog. La réécriture de ces

transformations dans une approche plus classique des transformations avec QVT, ATL ou Kermeta est une perspective possible à ce travail pour favoriser sa diffusion.

### 1.3.3 Quelques propriétés de l'opération de composition

Dans [HKR+07], les auteurs citent différentes propriétés des compositions de modèles définies sur les mêmes métamodèles. Nous les reprenons ici et les discutons au regard de notre problématique générale, la composition d'activités. Leur description formelle qui soulève plusieurs problèmes sera détaillée dans la partie I de ce document. Leur exploitation sera plus détaillée au niveau des applications dans la partie II.

**Préservation des propriétés :** La préservation partielle ou totale des propriétés est une des propriétés le plus souvent citée [HKR+07, BBF+06]. Même si la notion même de propriété d'une activité reste à préciser, nous pouvons par exemple nous poser la question : veut-on toujours lorsque l'on compose deux activités que la nouvelle activité fasse à référence à l'ensemble des actions référencées par chacune ? Inversement, la composition de deux activités peut-elle engendrer de nouvelles actions ? L'opérateur de composition peut-il correspondre à du raffinement ?

Nous définissons cette propriété dans un contexte général puis montrons au travers des applications les limites imposées par son respect et la difficulté à identifier des activités équivalentes.

**Idempotence** La propriété d'idempotence permet d'assurer que en composant une activités plusieurs fois, celle-ci ne sera pas dupliquée. Nous nous intéressons en particulier à caractériser les activités idempotentes (jouant le rôle d'élément neutre) lorsque l'opération de composition n'est pas elle-même idempotente.

**Commutativité, Associativité** Lorsque différents développeurs composent des modèles, nous recherchons à assurer une indépendance vis-à-vis de l'ordre de ces compositions, ce qui se traduit par la commutativité au sens où l'ordre des modèles au sein d'une composition n'intervient pas et plus généralement l'associativité, lorsque les interventions sont séquentielles.

Cette propriété est particulièrement recherchée dans le cadre de la programmation par aspects. En effet, lorsque le programmeur doit avoir connaissance des aspects précédemment appliqués pour appliquer un nouvel aspect, l'évolution devient vraiment difficile [LHBL06, FF05, TBB04].

Appliquer cette propriété à l'intégration des codes à la manière des aspects revient à rechercher un langage de définition des aspects dont le tissage des aspects est indépendant de l'ordre d'application des aspects.

## 1.4 Rappel du plan

Dans ce contexte et en suivant la démarche énoncée précédemment, nous présentons maintenant nos travaux selon le plan suivant.

Dans la partie suivante, nous proposons une modélisation des activités (cf. §2), leur composition (cf. §3) et les propriétés (cf. §4) qui peuvent être attendues de la composition indépendantes d'un domaine d'application donné. Cette partie est conclue par un

résumé des propriétés que doivent vérifier chacune des fonctions dépendantes domaines qui composent l'algorithme (cf. §5).

La troisième partie présente deux interprétations de cette métacomposition des activités et leurs propriétés : la composition d'interactions entre composants hétérogènes (cf. §6), la composition d'orchestrations de services (cf. §7).

Enfin une dernière partie conclut ce travail par nos perspectives. L'annexe A décrit nos résultats (publications et logiciels) et les encadrements qui ont jalonnés ces travaux.

*Première partie*

# MODÉLISATION DE LA COMPOSITION D'ACTIVITÉS

## Dans cette partie...

## Contexte de recherche et contributeurs

*Ce travail a été épaulé par Claudine Peyrat qui a su m'éclairer dans les dédales du monde des graphes et des formalisations et les remarques de Paul Franchi qui m'a rappelé avec tant d'enthousiasme que les algorithmes pouvaient se définir indépendamment des domaines, l'essentiel était de savoir les "interpréter" [BFFN05, BFFZ06].*

Dans l'introduction nous avons établi que la composition d'activités était nécessaire à différents domaines d'applications, mais qu'elle se déclinait différemment en fonction des attentes associées à ces compositions. Notre objectif est d'établir les bases d'une composition d'activités qui s'expose indépendamment des domaines applicatifs, mais puisse être interprétée différemment en fonction des domaines. Sur cette base, nous exprimons les propriétés attendues telles que la validité des activités, l'associativité de la composition ou la préservation des propriétés par composition. Cette partie est consacrée à établir les bases de cet algorithme. La partie suivante démontre que au moins sur quelques domaines, ce n'est pas une utopie.

Dans cette partie, nous commençons par proposer une modélisation et une formalisation des activités (cf. §2).
Cette base nous sert à définir un algorithme de composition basée sur la fusion d'activités "pivots" (cf. §3) [10]. La fusion d'activités pivots peut créer de nouvelles activités ou au contraire en détruire. Les autres activités vont se composer en fonction de la fusion des activités pivots.
Nous caractérisons alors quelques propriétés (cf. §4) que l'on peut attendre de la composition d'activités.
Le chapitre 5 conclut cette partie en synthétisant les éléments nécessaire à la définition de la composition d'activités dans un domaine applicatif.

10. Contrairement à la notion de fusion telle que définie par [BBF+06], nous ne forçons pas l'activité résultante à contenir tous les éléments dont elle est issue. Nous choisissons néanmoins le terme de fusion car à partir de plusieurs activités, nous obtenons de nouvelles activités différentes.

# CHAPITRE 2

# Modélisation et formalisation des activités

Afin d'établir les bases de la composition des activités indépendamment des domaines applicatifs, nous formalisons à présent la notion d'activité avec un objectif de composition.

Pour cela, nous commençons par présenter un exemple de domaine qui illustre cette partie (cf. §2.1). Nous proposons ensuite une modélisation des activités (cf. §2.2) que nous formalisons au chapitre suivant (cf. §2.3).

## 2.1 Exemple : Domaine fil rouge

Pour faciliter la lecture de cette partie, nous basons nos exemples sur une sous-partie de l'application de composition des interactions entre composants hétérogènes décrite au chapitre 6.

Dans ce domaine, l'objectif est de contrôler les messages reçus par des composants hétérogènes. Plusieurs utilisateurs sont amenés à définir différents contrôles. Lorsque ces contrôles interviennent sur une même opération, il faut les composer. Nous considérons ces contrôles comme des activités qu'il s'agit donc de composer. Le caractère multi-utilisateurs de cette application nous conduit à rechercher l'associativité dans les compositions.

Un contrôle est défini comme une activité composite. Dans ce domaine dit *Fil Rouge*, une activité composite est constituée d'un ensemble d'activités simples correspondant à des envois de messages synchrones, à des vérifications de prédicats ou à l'exécution d'un message. Nous considérons que ces activités d'exécution d'un message constituent des pivots à l'instar d'un point de coupe autour de la réception des messages dans un langage d'aspect tel que AspectJ. Elles correspondent alors à un "*proceed*". La composition opère par fusion de ces activités. La fusion consiste à unifier les variables et à reporter l'ensemble des priorités et des conditions portant sur les activités fusionnées sur une nouvelle activité.

Par exemple, si nous notons `doi` un envoi de message, `proceed` l'exécution d'un message, `ci` la vérification d'un prédicat, "`if .. else ...`" une exécution conditionnelle, "*;*" la séquence et "`//`" l'absence d'ordre, voici deux exemples de compositions.
Si nous composons `1) do11;proceed;do12` avec `2) do21;proceed;do22`, nous attendrons comme résultat : `(do11//do21);proceed;(do12//do22)`

De même, si nous composons `3) if c1 proceed else do1` avec `4) if c2 (do41; proceed) else do2`, nous attendrons comme résultat en terme d'exécution :

```
if (c1 & c2) (do41; proceed)
if (not c1& c2) (do41 // do1)
if (c1 & not c2) do2)
if (not c1 & not c2) do1//do2
```

La non-utilisation de `proceed` dans une branche exprime que le comportement initial ne doit pas être exécuté.

Cet exemple est très simple. Il sert à poser l'ensemble des éléments de formalisation. De manière plus générale, la composition d'activités n'engendre pas forcément une seule activité mais un ensemble d'activités.

## 2.2 Modélisation

Nous posons ici la définition des actions et des activités qui font le cœur de notre travail. Par rapport à la définition de ces concepts dans UML 2.0, nous en proposons une définition simplifiée. L'intention sous-jacente à cette modélisation étant la composition, nous ne prenons en compte ni les flots de données, ni la gestion de signaux.

A l'instar d'UML Superstructure [OMG07], une **action** est l'unité fondamentale de spécification du comportement. Elle prend en entrée un ensemble de données et le convertit en un ensemble de sorties [1]. Une action représente un comportement élémentaire, elle ne peut pas être décomposée (cf. p327, [OMG07]). Une action fait référence à une **information** qui définit ce comportement.

Nous définissons une **activité** comme la spécification de la coordination de l'exécution d'un ensemble de comportements subordonnés.

Nous distinguons les *activités simples* qui font référence à une action, des *activités composites* [2] qui font référence à des ensembles d'activités simples. Parmi les activités simples, les *activités de test* (*TestActivity*) font référence à une action correspondant à la vérification d'un prédicat qui a donc une seule valeur de retour booléenne. La variable de sortie d'une activité de test ne peut pas être utilisée plusieurs fois comme une variable de sortie. Elle correspond à l'unique évaluation de l'action prédicat *(single-assignment rule [McG82])*. La manière dont les prédicats sont évalués est dépendante de la plateforme. Nous posons pour hypothèse que *H1 : l'évaluation des prédicats est sans effet de bord sur le système* [3].

L'exécution d'une activité peut être assujettie à la fin de l'exécution d'autres activités *(precededBy.incoming)* et à la vérification de conditions (*guard*). Les précédences *(precedence)* et conditions *(conditions)* établissent une relation d'ordre partiel entre les ac-

1. An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty.

2. car composée d'activités différentes.

3. Cette contrainte est nécessaire à assurer l'équivalence des résultats de compositions d'activités quelque soit l'ordre des compositions. Bien que la spécification UML 2.0 autorise les effets de bord dans les tests, elle le déconseille [OMG07, page 323].

tivités. Une activité précédée d'une autre activité ou gardée par une condition ne pourra s'exécuter que si l'activité précédente ou l'activité de test associée à la condition a terminé son exécution, l'inverse n'est pas vraie : l'exécution de l'activité précédente n'impose pas que la suivante soit exécutée. Toutes les activités qui sont assujetties à la fin d'exécution d'une activité $a$ sont donc contraintes par au moins les mêmes conditions que celles qui portent sur $a$.

La figure 2.1 résume notre modélisation du concept d'activités, elle correspond à la définition du MétaModèle dédié à la Composition d'Activités (MM4CA). La section 2.3, au travers de la formalisation, explicite les contraintes que nous posons sur ce modèle.

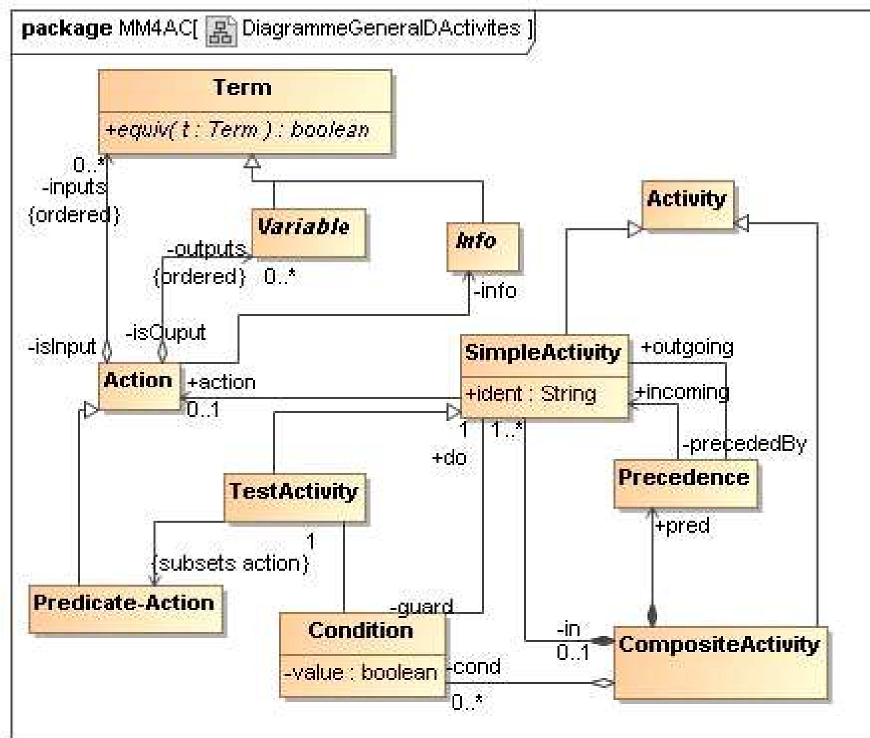



FIGURE 2.1: NOTRE MODÉLISATION DU CONCEPT D'ACTIVITÉ : LE MÉTAMODÈLE MM4CA

### 2.2.1 Simplifications vis-à-vis d'UML

Le flot de données, en particulier le passage d'une donnée en entrée d'activité à une action est donc simplifiée dans notre modélisation. Nous ne modélisons pas les flots de données, mais les déduisons des actions et des relations entre activités.

La figure 2.2 donne un exemple extrait de [Boc03] de modélisation des activités en UML. Elle présente à la fois a) la vue code b) la vue graphique en utilisation la notation UML des activités c) la vue "instance" du métamodèle UML2. La figure 2.3 montre sur le même exemple, notre modélisation en tant qu'"instance" du métamodèle MM4CA et la représentation graphique de l'activité composite.

a) Code textuel

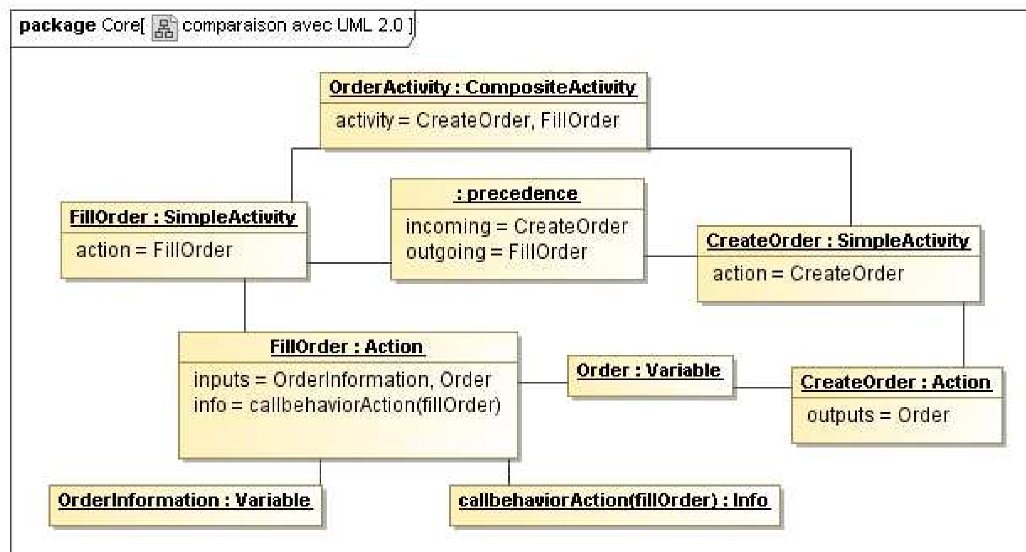
```
Order o;
o = new Order;
FillOrder(o);
```

b) Diagramme d'activités

c) Modélisation partielle en UML 2.0

FIGURE 2.2: EXEMPLE D'ACTIVITÉS EN UML 2.0 (EXTRAIT DE [BOC03])

a) Modélisation partielle relativement à notre métamodèle




b) Notre représentation graphique des activités

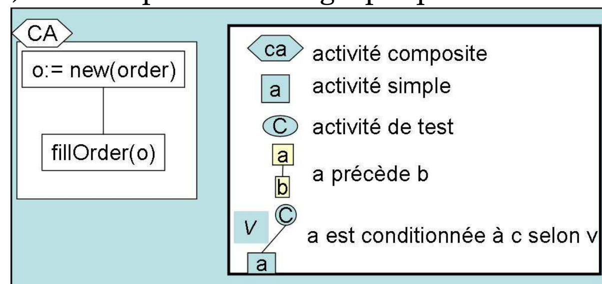



FIGURE 2.3: EXEMPLE DE [BOC03] MODÉLISÉ CONFORMÉMENT AU MÉTAMODÈLE MM4CA

### 2.2.2 Sémantique d'exécution d'une activité composite

Afin de clarifier la modélisation des activités, nous précisons la sémantique d'exécution des activités composites. Notre objectif n'est pas ici de définir un modèle exécutable des activités mais de préciser la sémantique associée à la modélisation.

Lors de l'exécution d'une activité composite, toutes les activités qui la composent sont activées et les variables créées. L'activité composite termine son exécution lorsque toutes les activités qui la composent terminent leur exécution (cf. figure 2.4).

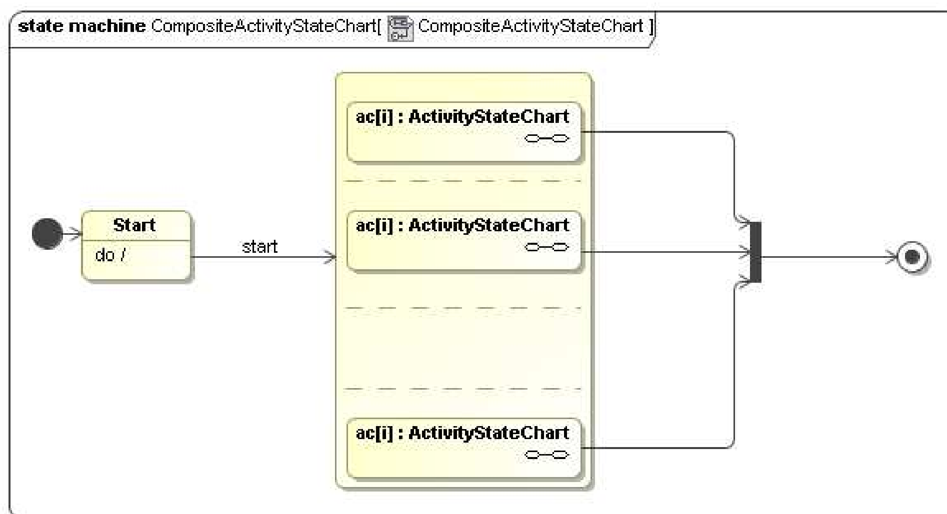



FIGURE 2.4: CYCLE DE VIE D'UNE ACTIVITÉ COMPOSITE

Une activité simple ne sera exécutée que lorsque toutes les activités qui la précèdent auront terminé leur exécution et les conditions ont toutes été valuées conformément aux valeurs attendues, ce qui implique que les activités de tests ont terminé leur exécution. Si une condition n'est pas vérifiée, l'activité est détruite puisque son exécution n'est plus nécessaire(cf. figure 2.5).

Notons que la fin d'exécution d'une activité est donc toujours visible via la génération de l'événement qui lui correspond.

Lorsqu'une activité de test est exécutée, sa variable de sortie prend la valeur booléenne correspondant à l'évaluation du prédicat. Cette variable ne pourra plus jamais être accédée en écriture. Toutes les conditions mettant en jeu cette activité de test porteront donc sur la même valeur.

Nous reprenons ici l'exemple extrait d'UML et en donnons une correspondance au niveau de l'exécution dans le langage $\Pi - diapason$ [Pou07], qui correspond à notre modélisation. La première activité simple *createOrder* correspond au processus suivant :

```
sequence (instanciate (invoke (
      operation (value('createOrder')),
      request_values(array([])),
      response_value(_order))),
   sequence (send(connection('endCreateOrder')),
   terminate))
```

La seconde activité doit attendre la fin de *createOrder* pour s'exécuter :

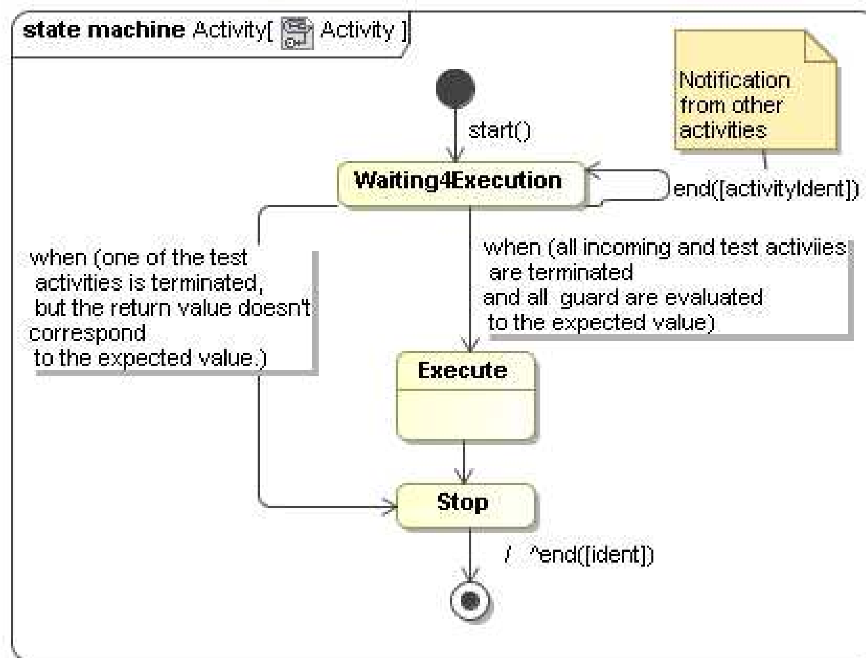



FIGURE 2.5: CYCLE DE VIE D'UNE ACTIVITÉ SIMPLE

```
sequence (instanciate (synchronize(
    connections(array([connection('endCreateOrder')])))),
sequence (instanciate (invoke (
    operation (value('FillOrder')),
    request_values(array([_order,_orderInformation]))
    *response_value(_))),*
sequence (send(connection('endFillOrder')),
terminate))
```

Enfin le comportement de l'activité composite peut être représenté comme :

```
sequence (instanciate (synchronize(
    connections(array([connection('endCreateOrder'),connection('endFillOrder')])))),
    terminate))
```

Nous pouvons sur cette base établir la sémantique d'une activité composite comme l'ensemble des traces d'exécution valides correspondantes. A la manière de Herrmann dans [HKR+07], nous notons *sm* la fonction qui associe à une activité composite sa sémantique sous la forme d'une trace d'exécution *(EventStream)* que nous limitons comme Gaaloul aux évènements signalant la fin d'exécution d'une activité simple notée ci-après par l'indentifiant de l'activité [GBG05].

**Exemple**

La seule trace valide correspondant à l'activité composite *OrderActivity* de la figure 2.3 est : <end(CreateOrder),end(FillOrder)>.
sm(OrderActivity)={<end(CreateOrder),end(FillOrder)>}

### 2.2.3 Domaines d'application

La définition des activités dépend des domaines d'application. En effet selon les domaines, les informations relatives aux activités sont différentes, de même que les compositions.
Un domaine se définit donc par un métamodèle qui raffine le métamodèle MM4CA, en particulier en précisant les concepts d'informations et de variables [4]. Ainsi la définition d'une variable pourra correspondre à un nom, à un couple (nom et type) ou être étendue pour prendre en compte la visibilité de la variable [Nan04]. L'ensemble des informations peut correspondre à des termes clos stipulant l'envoi de message dans les interactions entre composants, l'invocation de services dans une orchestration ou plus généralement des prédicats.
La composition des activités repose sur des opérations de comparaison ; un prédicat d'équivalence entre les informations et les variables doit être défini sur chaque domaine.
En complément au métamodèle définissant le domaine, des contraintes peuvent être ajoutées. En effet, en fonction des domaines applicatifs, les activités se définissent relativement à des langages [5]. Un domaine de composition d'activités correspondra donc à la fois à la donnée des espaces caractérisant les activités et à un ensemble de contraintes sur la définition des activités, par exemple le nombre d'occurrences de certaines informations dans une activité composite etc.

**`Exemple Fil rouge:` Domaine d'application**

Dans le cadre de notre exemple fil rouge, les éléments du domaine sont définis par le métamodèle de la figure 2.6.
Une **variable** est caractérisée par son seul nom. Deux variables sont toujours équivalentes.
Les **informations** correspondent à des termes clos [a], exprimant soit un envoi de messages *invocation(Nom de l'opération)*, soit un appel au message initial *proceed*, soit l'appel à un prédicat *predicat(Nom du prédicat)*. Dans ce domaine, deux informations sont équivalentes si elles sont identiques.

Des contraintes additionnelles limitent la construction des activités composites en assurant le déterminisme des accès aux variables ou l'unicité d'une activité pivot par branche. Ces points sont précisés au §2.3.5.

a. terme sans variable.

## 2.3 Formalisation

A présent, nous formalisons les concepts modélisés. L'objectif est ici de définir une représentation intermédiaire des activités composites sur laquelle l'algorithme de compo-

4. Les fonctions définies au niveau du métamodèle MM4CA s'appliquent sur des modèles conformes à un même domaine d'application. Nous utilisons donc la covariance dans un cadre simplifié. Les exceptions sont explicitées sous la forme de conflits[Duc02].

5. symboles et règles utilisés pour communiquer

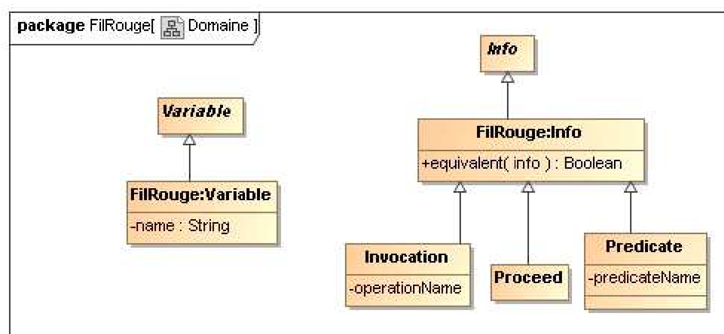



FIGURE 2.6: DOMAINE DE L'EXEMPLE FIL ROUGE

sition va être défini indépendamment du domaine d'application. L'adéquation entre cette représentation et un domaine applicatif sera établie par le raffinement du métamodèle MM4CA en un métamodèle $D$ dit *domaine d'interprétation* (cf. 2.2.3) et la définition d'un ensemble de fonctions sur ce domaine ; cet ensemble de fonctions est dit *interprétation*. Le domaine d'interprétation précise les éléments qui définissent les activités (variables et informations), ainsi que les contraintes propres au domaine cible (cf. 2.3.5).
Nous distinguons les fonctions et propriétés "libres" qui sont définies relativement au métamodèle MM4CA, des fonctions et propriétés sur domaine qui seront définies (resp. vérifiées) sur les domaines.
Nous précisons ces éléments à présent.

### 2.3.1 Domaines d'application : domaines d'interprétation

Comme nous l'avons vu au niveau de la modélisation, les variables et les informations (info) sont porteuses d'éléments différents en fonction des domaines d'application.
Soit $\mathcal{V}$, l'ensemble infini dénombrable des variables du domaine, sur lequel est défini une relation d'équivalence notée $\equiv$.
Soit $\mathcal{I}$, l'ensemble des informations du domaine, sur lequel est défini une relation d'équivalence notée $\equiv$.
Soit $\mathcal{ID}$, l'ensemble des identifiants.
Soit $\mathcal{T}$, l'ensemble des termes, $\mathcal{V} \cup \mathcal{I} \cup \mathcal{ID} \subset \mathcal{T}$. Une relation d'équivalence est définie sur les termes, notée $\equiv$.
Soit $\Upsilon$, l'ensemble des erreurs.
Un domaine d'interprétation correspond à la donnée des espaces caractérisant les activités et à un ensemble de contraintes sur la définition des activités, par exemple le nombre d'occurrences d'un type d'informations dans une activité composite. Ces points seront abordés lorsque nous aurons formalisé les activités (cf. 2.3.5).

### 2.3.2 Action

Une **action** est définie par un triplet $(info, inputs, outputs) \in (\mathcal{I}, \mathcal{T}^*, \mathcal{V}^*)$ où l'*info* représente les informations portées par l'action, *inputs* désigne la liste des termes en entrée et *outputs* la liste des variables en sortie. Chacune de ces listes peut être vide.
Nous notons $Action$, l'ensemble des actions.

Nous définissons les fonctions suivantes : $\forall a = (info, inputs, outputs) \in Action$,
***info*** : $info(a) = info$,
***inputs*** : $inputs(a) = inputs$,
***outputs*** : $outputs(a) = outputs$.

**`Exemple Fil rouge:` Représentation d'une action**

Soit l'action définie en pseudo-Java : *controler.notify('consumed',c)*
Une formalisation de cette action est :
`a = (invocation(notify),{controler,'consumed',c},{})` où
`info(a)=invocation(notify)`
`inputs(a)={controler,'consumed',c}` [a]
`outputs(a)={}`
Soit l'action définie en pseudo-Java : *result := proceed(x);*
Une formalisation de cette action est :
`a' = (proceed,{x},{result})` où
`info(a')=proceed`
`inputs(a')={x}`
`outputs(a')={result}`

*a*. Nous confondons pour simplifier l'identification des variables et leur nom.

### 2.3.3 Activité simple

Une **activité simple** est définie par un couple $(ident, action) \in \mathcal{ID} \times Action$ où *ident* identifie l'activité et *action* désigne une action.

Nous notons $\mathcal{SA}$ l'ensemble des activités simples.

Étant donnée une activité $a = (id, act)$,
nous définissons les fonctions associées suivantes :
$ident(a) = id$
$action(a) = act$
$inputs(a) = inputs(act)$
$outputs(a) = outputs(act)$
$variables(a) = outputs(a) \cup inputs(a) \cap \mathcal{V}$
Nous définissons la fonction *idents* qui renvoie la liste des identifiants d'un ensemble d'activités simples.

**Une activité de test (TestActivity)** est une activité simple dont l'action associée a une unique variable booléenne en sortie. Cette variable ne peut pas être accédée en écriture par une autre activité. Comme toutes les activités dans notre travail, une activité de test n'est exécutée qu'une fois. La variable en sortie représente la valeur résultante du test. Peu importe donc quand la valeur d'un test est utilisée, elle reste la même dès que l'activité de test a terminé son exécution. Ce point est important ; il différencie la notion d'activité de test, de la notion de garde qui ne porte pas en elle le moment de sa valuation et induit alors des problèmes de non-déterminisme [6].

6. L'évaluation d'une condition comme x<1 peut être vraie pour une valeur de x et devenir fausse plus tard lorsque la valeur de x est modifiée. Ces deux évaluations sont considérées comme des tests. Ainsi le

Nous notons $\mathcal{TA}$ l'ensemble des activités de tests : $\mathcal{TA} \subset \mathcal{SA}$

**Exemple Fil rouge: Activités**

Le code suivant est représenté graphiquement sous la forme d'activités par la figure 2.7.
***Activité composite ca1 :***

```
(a11)   size = x.size();
(t1)    if  (size>100)
(a12)       x.reduce();
(a13)       proceed(x);
        else
(a14)       result := proceed(x);
(a15)       memorize(result)
```

***Activité composite ca2 :***

```
(a21)   log('before');
(a22)   proceed(y);
(a12)   log('after')
```

***Activité composite ca3 :***

```
(t3)    if (x.isInColor())
(a31)       cb := ColorPrinter.capacity();
(a32)       proceed(z);
(a33)       ca := ColorPrinter.capacity();
(a34)       c=cb-ca;
(a35)       Controler.notify('consumed',c)
        else
(a36)       proceed(z);
```

Nous explicitons ici quelques-unes des différentes activités simples correspondantes :

```
(a11,(invocation(size),{x},{size})).
(a35,(invocation(notify),{controler,'consumed',c},{})).
(t1,(predicate(sup),{size,'100'},{V1})).
(a14,(proceed,{x},{result})).
```

Notons que la variable singleton `V1` ne sera jamais utilisée directement. Elle représente le résultat de l'évaluation du test. Elle n'est valuée qu'une fois.

### 2.3.4 Activité composite

Une **activité composite** est représentée par un graphe orienté $(La, R)$ où $La \in \mathcal{SA}^+$ est l'ensemble des activités simples constituant les sommets du graphe, et $R$ est une relation d'ordre stricte définissant les arcs du graphe, étiquetés dans $\{\varepsilon, true, false\}$ tel que :
$Lt \subseteq La, Lt \in \mathcal{TA}^+$ est l'ensemble des sommets définissant des activités de tests.
$R_p \subseteq R$ est l'ensemble des arcs étiquetés par $\varepsilon$.
$R_c \subseteq R$ est l'ensemble des arcs dans $Lt \times La$ étiquetés par $true$ ou $false$.

---

résultat de la première évaluation est mémorisée et toutes les activités qui sont conditionnées par ce test seront exécutées, même si la variable x est modifiée par d'autres activités. Une activité est conditionnée par le résultat d'un test pas par la vérification de la condition elle-même.

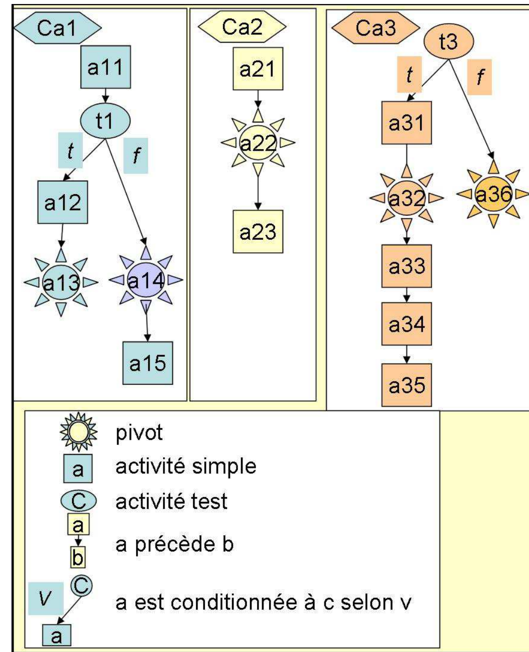



FIGURE 2.7: REPRÉSENTATION GRAPHIQUE DES ACTIVITÉS PRÉSENTÉES DANS L'EXEMPLE FIL ROUGE DE LA PAGE 36.

$R_p \cup R_c = R$.

Par simplification nous noterons en fonction des besoins une activité composite par un couple $(La, R)$ ou un triplet $(La, R_p, R_c)$.
Un arc $(a1, a2, x)$ dans $R$ exprime que l'activité $a1$ doit avoir terminé son exécution pour que l'activité $a2$ puisse être exécutée. Par voie de conséquence, la relation $R$ est **transitive**, une activité ne pouvant s'exécuter que si tous les prédécesseurs de ses prédécesseurs ont terminé leur exécution.
Nous faisons le choix dans une activité composite de ne représenter que les relations directes entre les activités et pas celles déduites par transitivité. Nous notons $R^*(a)$, l'ensemble des prédécesseur, même indirects, d'une activité $a$ relativement à la relation $R$.
Un arc $(a1, a2, value)$ dans $R_c$ exprime que l'activité $a1$ est une activité de test qui doit avoir terminé son exécution en retournant une valeur correspondant à $value$ pour que l'activité $a2$ puisse être exécutée.
Notre travail ne porte pas sur la détermination de cette valeur. Mais nous utilisons cette valeur pour calculer les compositions d'activités.

Nous notons $R_{|AL}$, la **restriction** d'une relation $R$ à un ensemble d'activités simples $AL$ :
$R_{|AL}AL = \{(a_i, a_j, e) \in R | a_i, a_j \in AL\}$

Nous notons *idents(R)* l'ensemble des identifiants des activités mises en jeu dans la relation.

Nous notons $\mathcal{CA}$ l'ensemble des activités composites.

Nous définissons les fonctions :
Soit $ac = (La, R_p, R_c) = (\{a_1, ...a_n\}, R), R = R_p \cup R_c$
$activities(ac) = La$,
$precedenceRelation(ac) = R_p$,
$conditionRelation(ac) = R_c$.
$variables(ac) = \cup_{i=1}^{i=n} variables(a_i)$

Nous nommons les relations $R_p$, ***relation de précédence*** et les relations $R_c$, ***relations de conditions***.

#### 2.3.4.1 Relation de précédence

Nous notons $\mathcal{R}_p$ l'ensemble des relations de précédence.
Nous notons $pred(R_p, a)$ (resp. *succ*) l'ensemble éventuellement vide des activités activités précédentes (resp. successives), relativement à la relation de précédence $R_p$.
Soit $R_p \in \mathcal{R}_p$,
$pred(R_p, a) = \{a_j | (a_j, a) \in R_p\}$,
$succ(R_p, a) = \{a_j | (a, a_j) \in R_p\}$

Nous étendons ces fonctions à la fermeture transitive de $R_p$, notées respectivement $pred^*$ et $succ^*$.

**`Exemple Fil rouge:` Relations de précédence**

Dans les exemples de la figure 2.7,
- la relation de précédence associée à *ca1* est :
  $Rp_1$= `{pred(a11,t1),pred(a12,a13),pred(a14,a15) }`
- la relation de précédence associée à *ca2* est :
  $Rp_2$= `{pred(a21,a22),pred(a22,a23) }`
- la relation de précédence associée à *ca3* est :
  $Rp_3$= `{pred(a31,a32),pred(a32,a33),pred(a33,a34),pred(a34,a35) }`

L'union des relations de précédence se définit par l'union ensembliste.

#### 2.3.4.2 Relation de condition

Nous notons $\mathcal{R}_c$ l'ensemble des relations de condition.
Nous notons $condTrue(R_c, a)$ (resp. $condFalse(R_c, a)$) l'ensemble éventuellement vide des activités de tests dont la valeur de retour doit être valuée à $true$ (resp. à $false$).
Soit $R_c \in \mathcal{R}_c$,
$condTrue(R_c, a) = \{a_i | (a_i, a, true) \in R_c\}$,
$condFalse(R_c, a) = \{a_i | (a_i, a, false) \in R_c\}$

Sur la base de ces deux fonctions, nous définissons la fonction *cond* qui à partir d'une activité simple renvoie l'ensemble éventuellement vide des activités de tests qui gardent cette activité.
Soit $R_c \in \mathcal{R}_c$,

$cond(R_c, id) = condTrue(R_c, id) \cup condFalse(R_c, id)$

**`Exemple Fil rouge:` Relations de condition**

Dans les exemples de la figure 2.7, toutes les relations n'ont pas été représentées, les voici :
- la relation de condition associée à *ca1* est :
  $Rc_1$= `{cond(t1,a13,true),cond(t1,a12,true),cond(t1,a14,false),cond(t1,a15,false) }`
- la relation de condition associée à *ca2* est :
  $Rc_2 = \emptyset$
- la relation de condition associée à *ca3* est :
  $Rc_3$= `{cond(t3,a36,false),cond(t3,a31,true), cond(t3,a32,true),cond(t3,a33,true), cond(t3,a34,true),cond(t3,a35,true) }`
  Notons que les activités simples d'identifiant *a32* et *a36* sont exclusives.

Nous définissons à présent les fonctions $condTrue^*$ et $condFalse^*$ qui pour une activité et des relations de précédence et de conditions données renvoient l'ensemble des conditions à laquelle cette activité doit être assujettie, i.e. l'union des conditions qui portent sur elle-même et sur ses prédécesseurs. En effet, si l'un de ses prédécesseurs ne devait jamais s'exécuter, elle ne s'exécuterait pas.

$condTrue^*(R_p, R_c, a) =$
$\quad condTrue(R_c, a) \bigcup \{cond_i | a_i \in pred^*(R_p, a), (cond_i, a_i, true) \in R_c\}$
$condFalse^*(R_p, R_c, id) =$
$\quad condFalse(R_c, a) \bigcup \{cond_i | id_i \in pred^*(R_p, a), (cond_i, a_i, false) \in R_c\}$

**Propriété 1 (Activités exclusives)** *Deux activités $a1$ et $a2$ sont **exclusives** au regard d'une relation, notée $exclusive(R, a1, a2)$ si :*
*Soient $id1 = ident(a1), id2 = ident(a2)$*
$(condTrue^*(R, a1) \cap condFalse^*(R, a2) \neq \emptyset) \vee (condTrue^*(R, a2) \cap condFalse^*(R, a1) \neq \emptyset)$

**`Exemple Fil rouge:` Activité composite**

Sur les exemples de la figure 2.7, les activités composites sont :

`ca1= ({a11,t1,a12,a13,a14,a15},`$Rp_1$`,`$Rc_1$`)`
`inputs(ca1)={x}`
`outputs(ca1)={result}`

`ca2= ({a21,a22,a23},`$Rp_2$`,`$Rc_2$`)`
`inputs(ca2)={y}`
`outputs(ca2)={result}`

`ca3= ({t3,a31,a32,a33,a34,a35,a36},`$Rp_3$`,`$Rc_3$`)`
`inputs(ca3)={z}`
`outputs(ca3)={result}`

**Propriété 2 (Activité composite valide versus incohérente)** *Une activité composite $ac = (La, R_p, R_c)$ est **valide** si les tests sur les activités ne s'excluent pas, dans le cas*

*contraire elle est dite* ***incohérente****.*
$\forall a_i \in La, condTrue^*(R_c, a_i) \bigcap condFalse^*(R_c, a_i) = \emptyset$

Les activités composites sur lesquelles nous travaillons sont des activités composites valides.

**`Exemple Fil rouge`: Activités composites valides versus incohérentes**

Dans tous les exemples donnés jusqu'à présents les activités composites sont valides.
Voici un exemple d'activité composite incohérente :
*ca*=`({a1,t1},{},{cond(t1,a1,false),cond(t1,a1,true)})`

**Fonction 1** (F1) : *Copie d'activité composite* La fonction de copie d'activités composites (*copy*) est définie de $\mathcal{CA} \longrightarrow \mathcal{CA}$.
Soit `ac` une activité composite, *ac'= copy(ac)* est une nouvelle activité composite dont toutes les activités qui la composent ont été recopiées (leur identifiant étant nouveau et leurs variables renommées), et les relations de précédence et de condition redéfinies sur ces nouvelles activités.

**Une branche** relative à une activité simple *a* dans une activité composite correspond à l'ensemble des activités qui sont conditionnées par au moins les mêmes conditions que l'activité *a*.

**Fonction 2** (F2) : *Branche d'une activité simple dans une activité composite*
La fonction *getBranch* est définie de $\mathcal{CA} \times \mathcal{SA} \rightarrow \mathcal{SA}^*$
Soit $ac = (La, R_p, R_c), a \in La, getBranch(ac, a) = \{a_i \in La |$
$condTrue^*(R_c, a) \subseteq condTrue^*(R_c, a_i), condFalse^*(R_c, a) \subseteq condFalse^*(R_c, a_i)\}$.
La branche correspondant à une activité simple non conditionnée dans une activité composite est l'ensemble des activités simples associées à l'activité composite.

### 2.3.5 Domaine d'application : contraintes

**Fonction sur domaine 1** (FD1) : *Appartenance d'une activité composite à un domaine*
Une activité composite définie relativement à un domaine appartient à ce domaine si elle respecte les contraintes imposées par ce dernier. Afin de permettre l'expression de ces contraintes indépendamment de tout formalisme, nous définissons la fonction $isInDomain_D$ définie sur $\mathcal{CA}$ qui renvoie vraie si l'activité composite respecte les contraintes du domaine et faux sinon.
La composition des activités composites appartenant à un domaine vise à construire des activités composites dans le même domaine.
Voici une propriété que les domaines étudiés visent à respecter.

**Propriété 3 (Activité composite non déterministe)** *Une activité composite valide est* ***non déterministe*** *s'il existe deux activités qui sont non exclusives et non ordonnées et qui accèdent à une même variable, dont une fois en écriture :*
*Soit* $ac = (La, R) = (La, R_p, R_c)$*, ac est non déterministe si*
$\exists a_i \in La \wedge \exists a_j \in La, not(exclusive(R, a_i, a_j)) \wedge$
$a_i \notin R^*(a_j) \wedge a_j \notin R^*(a_i) \wedge$
$((outputs(a_j) \cap variables(a_i) \neq \emptyset) \vee ((outputs(a_i) \cap variables(a_j) \neq \emptyset)))$

**`Exemple Fil rouge:` Contraintes du Domaine**

Les contraintes relatives au domaine *FilRouge* sont :

1. dans une activité composite, il existe au plus une activité correspondant à une information *proceed* par branche (donc deux activités *proceed* dans une même activité composite sont exclusives) ;
2. une activité *proceed* ne peut avoir en entrée que des variables, la variable en sortie est toujours *result*, elle peut être implicite ; toutes les variables en entrée des activités pivots dans une même activité composite sont les mêmes ;
3. une activité composite est déterministe relativement à la propriété 3 ;
4. l'accès en lecture aux variables d'une activité composite n'est possible que si celles-ci ont précédemment été affectées ou correspondent aux variables de l'activité *proceed*[a].

La fonction $isInDomain_{FilRouge}$ vérifie que toutes les propriétés précédentes sont bien vérifiées.

a. En effet, ces variables désignant les paramètres de l'appel, elles ont donc une valeur.

Dans chaque domaine, différentes propriétés peuvent être définies que la fonction $isInDomain_D$ doit vérifier éventuellement par utilisation de transformations, par exemple vers des algèbres de processus [Pou07].

### 2.3.6 Système

Un **système** est un ensemble d'activités composites, identifiées deux à deux comme distinctes.
Un système est **valide** s'il existe au plus une activité composite ou simple pour un identifiant donné.
Soit $S$ un système défini par l'ensemble des activités composites $\{ac_1, ..ac_n\}$, nous définissons les fonctions d'accès suivantes :
$getCA_S : \mathcal{ID} \longrightarrow \mathcal{CA} \cup \Upsilon$ qui renvoie pour un identifiant donné l'activité composite correspondante si elle existe ou erreur.
$getSA_S : \mathcal{ID} \longrightarrow \mathcal{SA} \cup \Upsilon$ qui renvoie pour un identifiant donné l'activité simple correspondante si elle existe ou erreur.
$getSA^*_S : \mathcal{ID}^* \longrightarrow \mathcal{SA}^* \cup \Upsilon$ qui renvoie pour un suite d'identifiants donnée les activités simples correspondantes si toutes existent ou erreur.

Les fonctions de création des activités sont établies sur cette base. Les identifiants créés sont obligatoirement nouveaux au sein d'un système donné $S$.
$createCA_S : \mathcal{SA}^* \times \mathcal{R}_p \times \mathcal{R}_c \longrightarrow \mathcal{CA}$ crée à partir d'une liste d'activité simples et des relations associées, une nouvelle activité composite d'identifiant unique dans le système $S$.
$createSA_S : Action \longrightarrow \mathcal{SA}$ crée à partir d'une action, une nouvelle activité simple d'identifiant unique dans le système $S$.

Un système est bien **construit** si toute activité simple qui le compose appartient au plus à une activité composite.
Pour simplifier le texte qui suit nous omettrons le système qui est cependant sous-jacent aux différentes fonctions présentées ci-après : elles opèrent toujours au sein d'un système. En particulier, l'ensemble des fonctions et notations définies sur les activités

simples ou composites s'appliquent aux identifiants des activités en utilisant l'unicité des identifiants.

### 2.3.7 Substitutions

Cette partie s'inspire de Mark Stickel [Sti81].

*Composant de substitution*
Un composant de A est un couple ordonné (v,t) où v est une variable et t un terme. Un composant de substitution explicite l'affectation d'un terme à une variable.

*Substitution*
Une substitution $\sigma$ est un ensemble de composants de substitution avec des premiers éléments variables distincts. Appliquer une substitution à une expression revient à remplacer ces variables par les termes correspondants. Les composants de substitution sont appliqués en parallèle, et les occurrences de variable introduites par les termes ne sont pas remplacées même si la variable apparaît comme un premier terme d'un composant de substitution.

Le **domaine de la substitution** $\sigma$ (noté $dom(\sigma)$) est l'ensemble des variables $x$ telles que $\sigma(x) \neq x$.

Nous notons $\{(v_1, t_1)..,(v_n, t_n)\}$ la substitution de domaine $\{v_1, ..., v_n\}$ qui envoie chaque variable $v_i$ vers $t_i$, $1 \leq i \leq n$, où les $v_i$ sont supposées distinctes deux à deux. On note $\{\}$ la substitution dont le domaine est vide.

On appelle substitué d'un terme $t$ par une substitution $\sigma$, le terme obtenu par remplacement de toute occurrence $xi$ des variables dans $t$ appartenant à $dom(\sigma)$ par le terme substitué $\sigma(xi)$. Par extension, on note $\sigma(t)$ le substitué.
Étant donné un ensemble de termes $T = \{t1, ...tn\}$, on note $\sigma(T) = \{\sigma(t1), ...\sigma(tn)\}$

*Application d'une substitution*
Pour simplifier, nous notons $\sigma(a1)$ l'application d'une substitution $\sigma$ à une action $a1 = (info, inputs, outputs)$ où $\sigma(a1) = (\sigma(info), \sigma(inputs), \sigma(outputs))$.
L'application d'une substitution $\sigma$ à une activité simple $sa = (id, a)$ est notée $\sigma(sa)$, et $\sigma(sa) = (id, \sigma(a))$.

*Unification*
L'unification est un cas particulier de recherche de substitution, où étant donnés n termes $t_1, ...t_n$, nous recherchons la substitution $\sigma$ telle que $\sigma(t_1) = ... = \sigma(t_n)$. Si elle existe $\sigma$ est dit *unificateur* de $t_1, ...t_n$.
Lorsque n termes sont unifiables, on démontre qu'il existe un unificateur plus général unique au renommage des variables près.

### 2.3.8 Equivalences au sein d'un système

**Propriété 4 (Equivalence d'actions)** *Deux actions $a1$ et $a2$ sont équivalentes, s'il existe une substitution $\sigma$, telle que*
$\sigma(inputs(a1)) = \sigma(inputs(a2)) \wedge \sigma(outputs(a1)) = \sigma(outputs(a2)) \wedge \sigma(info(a1)) \equiv \sigma(info(a2))$
*Nous notons $a1 \equiv_\sigma a2$ ou plus simplement $a1 \equiv a2$*

Notons que cette équivalence repose sur l'équivalence des informations qui est dépendante du domaine.

**Propriété 5 (Equivalence d'activités simples)** *Deux activités simples sont équivalentes si leurs actions sont équivalentes.*

**Propriété 6 (Inclusion d'ensembles d'activités simples)** *Un ensemble $e1$ d'activités simples est inclus dans un ensemble $e2$ d'activités simples, si $\exists \sigma, \forall a_i \in e1, \exists a_j \in e2, \sigma(a_i) \equiv \sigma(a_j)$.*
*Nous notons cette inclusion : $e1 \subseteq_\sigma e2$, ou plus simplement $e1 \subseteq e2$.*

**Propriété 7 (Equivalence d'ensembles d'activités simples)** *Un ensemble $e1$ d'activités simples est équivalent à un ensemble $e2$ d'activités simples, si $\exists \sigma, e1 \subseteq_\sigma e2, e2 \subseteq_\sigma e1$.*
*Nous notons cette équivalence : $e1 \equiv_\sigma e2$, ou plus simplement $e1 \equiv e2$.*

**Propriété 8 (Inclusion de relations de précédence)** *Nous considérons qu'une relation de précédence $R_p$ est incluse dans une relation de précédence $R'_p$ si toutes les activités apparaissant dans la relation de précédence $R'_p$ sont soumises à au moins les précédences imposées par $R_p$, éventuellement indirectement, soit :*
*$R_p \subseteq_\sigma R'_p$ si $\forall (a_1, a_2, \varepsilon) \in R_p, \exists a'_1 \equiv_\sigma a_1, \exists a'_2 \equiv_\sigma a_2, a'_1 \in pred^*(R'_p, a'_2)$.*

**Propriété 9 (Inclusion de relations de condition)** *Nous considérons qu'une relation de condition $R_c$ est incluse dans une autre relation de condition $R'_c$ si toutes les activités apparaissant dans la relation de condition $R'_c$ sont soumises à au moins les conditions imposées par $R_c$, soit :*
*$R_c \subseteq_\sigma R'_c$ si*
*$\forall (a_1, a_2, true) \in R_c, \exists a'_1 \equiv_\sigma a_1, \exists a'_2 \equiv_\sigma a_2, a'_1 \in condTrue^*(R'_c, a'_2)$, et*
*$\forall (a_1, a_2, false) \in R_c, \exists a'_1 \equiv_\sigma a_1, \exists a'_2 \equiv_\sigma a_2, a'_1 \in condFalse^*(R'_c, a'_2)$.*

**Propriété 10 (Equivalence de relations de condition)** *$R_c \equiv_\sigma R'_c$ si $R_c \subseteq_\sigma R'_c \wedge R'_c \subseteq_\sigma R_c$.*

**Fonction 3** (F3) : *Restriction des ensembles par l'équivalence*

$AL_{|_\sigma NAL} = AL'$ tel que $AL' \subseteq AL, AL' \subseteq_\sigma NAL, \forall x \in (AL_{|_{AL'}}), x \notin_{sigma} NAL$
Les fonctions de restriction sur les relations de précédence et de condition sont définies à $\sigma$ près en utilisant l'équivalence.

**Propriété 11 (Activités simples dupliquées dans une activité composite)** *Nous dirons qu'une activité composite $ca = (La, R_p, R_c)$ contient des activités dupliquées, si :*
*$\exists sa_1 \in activities(ca), sa_2 \in activities(ca), sa_1 \equiv_\sigma sa_2, sa_1 \neq sa_2$ et*
*$pred(R_p, sa_1) \equiv_\sigma pred(R_p, sa_2)$ et*
*$succ(R_p, sa_1)) \equiv_\sigma succ(R_p, sa_2)$ et*
*$condTrue(R_p, sa_1)) \equiv_\sigma condTrue(R_p, sa_2)$ et*
*$condFalse(R_p, sa_1) \equiv_\sigma condFalse(R_p, sa_2))$.*
*Nous notons $sa_1 \equiv_{(La, R_p, R_c)} sa_2$ ou $sa_1 \equiv_{ca} sa_2$*

Notons qu'une activité composite $ca$ peut contenir des activités équivalentes ($\equiv$) sans que qu'elles soient dupliquées, puisque la duplication dans une activité composite ($\equiv_{ca}$) met en jeu également les relations.

**`Exemple Fil rouge:` activités dupliquées**

Soit l'activité composite `ca` :
`({(a,invocation(size),{x},{size}), (b,invocation(size),{y},{size}), (c,invocation(reduce),{x},{})},{pred(a,c)},{})`
Les activités a et b sont équivalentes $a \equiv b$ mais ne sont pas équivalente dans $ca$ : $a \not\equiv_{ca} b$, en effet $a$ précède $c$ ce qui n'est pas le cas de $b$.

**Propriété 12 (Inclusion d'activités composites)** *Une activité composite $ca1$ est incluse dans une activité composite $ca2$, $ca1 \subseteq_\sigma ca2$ si :*
$activities(ca1) \subseteq_\sigma activities(ca2) \wedge$
$precedenceRelation(ca1) \subseteq_\sigma precedenceRelation(ca2) \wedge$
$conditionRelation(ca1) \subseteq_\sigma conditionRelation(ca2)$.

**Propriété 13 (Equivalence d'activités composites)** *Une activité composite $ca1$ est équivalente à une activité composite $ca2$, si*
$ca1 \equiv_\sigma ca2 \iff (ca1 \subseteq_\sigma ca2) \wedge (ca2 \subseteq_\sigma ca1)$

Par définition de la fonction de copie, nous avons la propriété suivante sur les copies d'activités composites.

**Propriété 14 (Equivalence des copies d'activités composites)** $ac' = copy(ac) \Longrightarrow ac \equiv_\sigma ac'$

### 2.3.9 Normalisation d'une activité composite

Pour pouvoir composer les activités, nous définissons une opération de "normalisation" d'une activité composite en une activité composite valide afin qu'elle ne contienne donc pas d'activités en attente de deux activités qui s'excluent (cf. propriété 2). Nous faisons le choix de dupliquer les activités dans un tel cas et de mettre à jour les relations de précédence et de conditions.

Ainsi, par exemple, *(if (c1) do1 else do2) ; do3*, sera réécrit pour correspondre à : *if (c1) (do1 ; do3a) else (do2 ; do3b)*. Notons que la première écriture ne peut pas correspondre à une activité composite valide puisque en fonction de sa représentation dans notre modèle : soit *do3* est conditionnée par des tests qui s'excluent, soit elle n'est pas soumise à l'union des tests de ses prédécesseurs.
Cette opération de transformation est appelée **normalisation** et est définie comme suit.

**Fonction 4** (F4) : *Normalisation*
La fonction de normalisation est définie de $\mathcal{SA}^+ \times \mathcal{R}_p \times \mathcal{R}_c \longrightarrow \mathcal{SA}^+ \times \mathcal{R}_p \times \mathcal{R}_c$.
soit $(la, R_p, R_c)$
$normalize(la, R_p, R_c) = (la', R_p', R_c')$ telle que :
$la \equiv_\sigma la', R_p \equiv_\sigma R_p', R_c \equiv_\sigma R_c'$ et
$\forall a_i \in La', \forall a_j \in pred*(R_p', a_i)$,

$condTrue(R'_c, a_j) \subseteq_\sigma condTrue(R'_c, a_i), condFalse(R'_c, a_j) \subseteq_\sigma condFalse(R'_c, a_i)$.

Nous définissons la fonction de normalisation d'une activité composite de $\mathcal{CA} \longrightarrow \mathcal{CA}$, comme : $normalize(ca) = ca', ca = (la, R_p, R_c), normalize(la, R_p, R_c) = (la', R'_p, R'_c), ca' = (la', R'_p, R'_c)$.

Par définition nous avons les propriétés suivantes.

**Propriété 15 (Propriétés de la normalisation)** $\forall ca \in \mathcal{CA} normalize(ca) = ca_n \Longrightarrow \exists \sigma$ *telle que :*
$activities(ca_n) \equiv_\sigma activities(ca)$
$precedenceRelation(ca_n) \subseteq_\sigma precedenceRelation(ca)$
$conditionRelation(ca_n) \subseteq_\sigma conditionRelation(ca)$
*et*
$\forall a \in activities(ca), \exists \{a_1...a_p\} \subseteq activities(ca_n), \forall i, a_i \equiv_\sigma a$
$precedenceRelation(ca_n) = R_{p_n}, conditionRelation(ca_n) = R_{c_n},$
$\bigcup_{i=1}^{i=p} pred(R_{p_n}, a_i) \equiv_\sigma pred(R_p, a)$
$\bigcup_{i=1}^{i=p} condTrue(R_{c_n}, a_i) \equiv_\sigma condTrue(R_c, a)$
$\bigcup_{i=1}^{i=p} condFalse(R_{c_n}, a_i) \equiv_\sigma condFalse(R_c, a)$

La figure 2.8 visualise un exemple de normalisation.

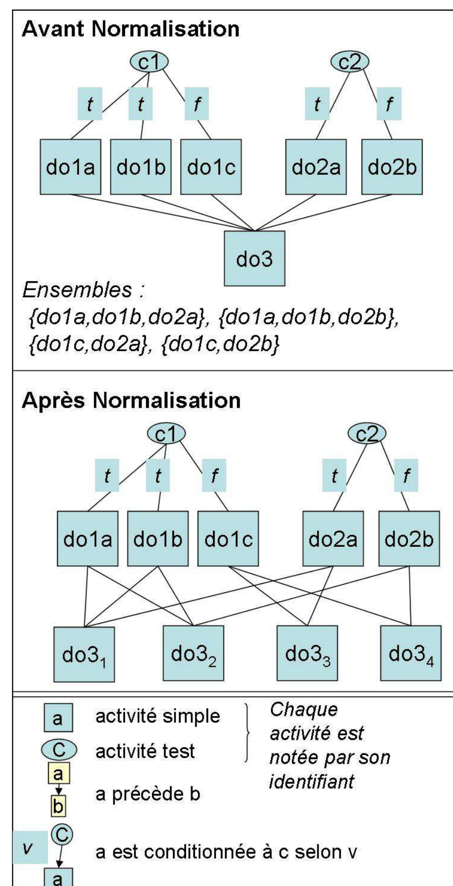
**Propriété 16 (Normalisation et duplication)** $\forall ca \in \mathcal{CA}$*, Si elle ne comporte pas d'activités équivalentes alors après normalisation, elle ne contient pas d'activités dupliquées, Si elle ne comporte pas d'activités dupliquées, la normalisation n'introduit pas de duplication entre les activités normalisées.*

Lorsqu'une activité est "dupliquée" pendant la normalisation, les nouvelles activités ont pour prédécesseur seulement une partie des prédécesseurs de l'activité remplacée. En conséquence, ces activités sont équivalentes entre elles mais non dupliquées au sein de l'activité composite. Une nouvelle activité peut cependant être équivalente à une autre activité préexistante.

### 2.3.10 Normalisation sur domaine

Afin de rendre valide une activité composite sur son domaine d'application, nous introduisons une fonction de normalisation sur domaine qui permet soit d'invalider un ensemble d'activités simples liées par une relation de précédence et de condition comme n'appartenant pas au domaine (dans ce cas, elle correspond simplement à la fonction 1 $isInDomain_D$), soit de corriger cet ensemble.

**Fonction sur domaine 2** (FD2) : *Normalisation Sur Domaine*
La fonction de normalisation sur domaine est définie de :
$\mathcal{SA}^+ \times \mathcal{R}_p \times \mathcal{R}_c \longrightarrow (\mathcal{SA}^+ \times \mathcal{R}_p \times \mathcal{R}_c) \vee \Upsilon$.
Soit $(la, R_p, R_c), normalizeOnDomain_D(la, R_p, R_c) = (la', R'_p, R'_c)$ si la normalisation sur domaine est possible ; elle lève sinon l'erreur "impossible normalisation sur domaine".




L'activité $do3$ est initialement conditionnée par des activités qui s'excluent. L'activité est alors dupliquée en autant d'activités que de traces possibles. Par exemple $do3_1$ représente l'activité $do3$ lorsque les tests $c1$ et $c2$ sont évalués à vrai. Par exemple $do3_4$ représente l'activité $do3$ lorsque les tests $c1$ et $c2$ sont évalués à faux.

FIGURE 2.8: EXEMPLE DE NORMALISATION

**`Exemple Fil rouge:` Normalisation sur Domaine**

La fonction de normalisation sur domaine $normalizeOnDomain_{FilRouge}$ consiste uniquement à vérifier que les activités respectent les propriétés du domaine. Elle correspond donc si l'appel à la fonction $isInDomain_{FilRouge}$ renvoie vraie à retourner les ensembles sans modification.

C'est au niveau de cette fonction que nous pouvons trouver des transformations vers d'autres espaces technologiques et en particulier les algèbres de processus afin de valider des propriétés qu'il serait difficile de vérifier au niveau de notre formalisation. Par exemple, dans [Pou07], le langage diapason-* permet de formuler des propriétés spécifiques à une orchestration qu'il pourrait être intéressant de valider par une transformation vers ce langage. De même le respect de contraintes de temps pourrait être situé au niveau de cette fonction. Néanmoins des propriétés telles que l'unicité d'une invocation (une et une seule facture est produite), ou la présence d'une séquence d'activités (il n'y a pas d'action de facturation qui ne soit suivie d'une livraison) semble pouvoir être vérifiées directement dans notre représentation, et ceci d'autant plus facilement que nous utilisons Prolog pour représenter les graphes d'activités.

# CHAPITRE 3

# Composition d'activités

La composition d'activités vise à définir une nouvelle activité composite à partir d'un ensemble d'activités composites. Contrairement aux travaux de Straeten [VDS05], la composition d'activités ne repose pas nécessairement sur la préservation du comportement comme dans le cas d'un raffinement. En effet lors de la composition il est possible de rendre obsolète un ancien comportement (par exemple parce que *deprecated* ou devenu interdit lorsque l'on ajoute de la sécurité). Ainsi la composition d'activités nécessite des mécanismes de "fusion" dans lesquels des activités se fusionnent pour engendrer de nouvelles activités (par exemple, la fusion de deux invocations à un même service peut donner lieu à une nouvelle invocation dans laquelle les valeurs des paramètres ont été agrégées par appel à une nouvelle activité). La maîtrise du mécanisme de composition est l'objectif de ce chapitre. Pour ce faire, nous précisons l'algorithme que nous proposons et les propriétés qui doivent être vérifiées ou peuvent être attendues et leurs conséquences sur les différentes étapes de l'algorithme.
Dans la partie II *applications* nous donnons différentes interpretations correspondantes à cet algorithme.
Comme dans le chapitre précédent pour faciliter la lecture, nous illustrons l'algorithme sur le domaine fil rouge. Avec le même objectif, nous donnons à présent une description informelle de l'algorithme de composition.

## 3.1 Vue synthétique de la composition d'activités composites

### Procédé de composition : présentation informelle

La composition d'un ensemble d'activités composites indépendantes va dépendre du domaine d'application. Cependant nous avons dégagé un algorithme général qui repose sur les étapes suivantes :

1. Copie des activités composites à composer.
   Cette étape nous permet de préserver l'unicité des activités simples relativement à une activité composite, de même que la portée des variables relativement à chaque activité composite.

2. Recherche des activités fusionnables (cf. 3.2).
   Entre plusieurs activités composites seules certaines activités simples sont potentiellement fusionnables, elles sont dites pivots. Les activités pivots fusionnables deux à deux vont constituer des paires de fusion, puis par regroupement des ensembles de fusion.
   Cette étape est à rapprocher de la recherche de correspondances telle que définie dans [Pas06b].

   La fonction $detectMergePairs_D$ dépendante du domaine capture cette étape, tandis que la fonction $detectMergeSets$ sur la base de la fonction précédente calcule les ensembles de fusion.

   A cette étape des **conflits** dit de "**pivots**" pourront être détectés lorsque des fusions attendues ne peuvent pas être réalisées.
3. Calcul des nouvelles activités et relations engendrées par la fusion d'ensembles d'activités pivots et détermination des transformations à opérer sur l'union des activités simples composant les activités composites initiales (cf. 3.4).
   La fusion d'un ensemble d'activités simples peut donner lieu à la création de nouvelles activités, l'ajout de nouvelles précédences et conditions mettant en jeu ces nouvelles activités et un ensemble de transformations à opérer.

   La fonction $mergeSimpleActivities_D$, dépendante domaine, capture cette étape.

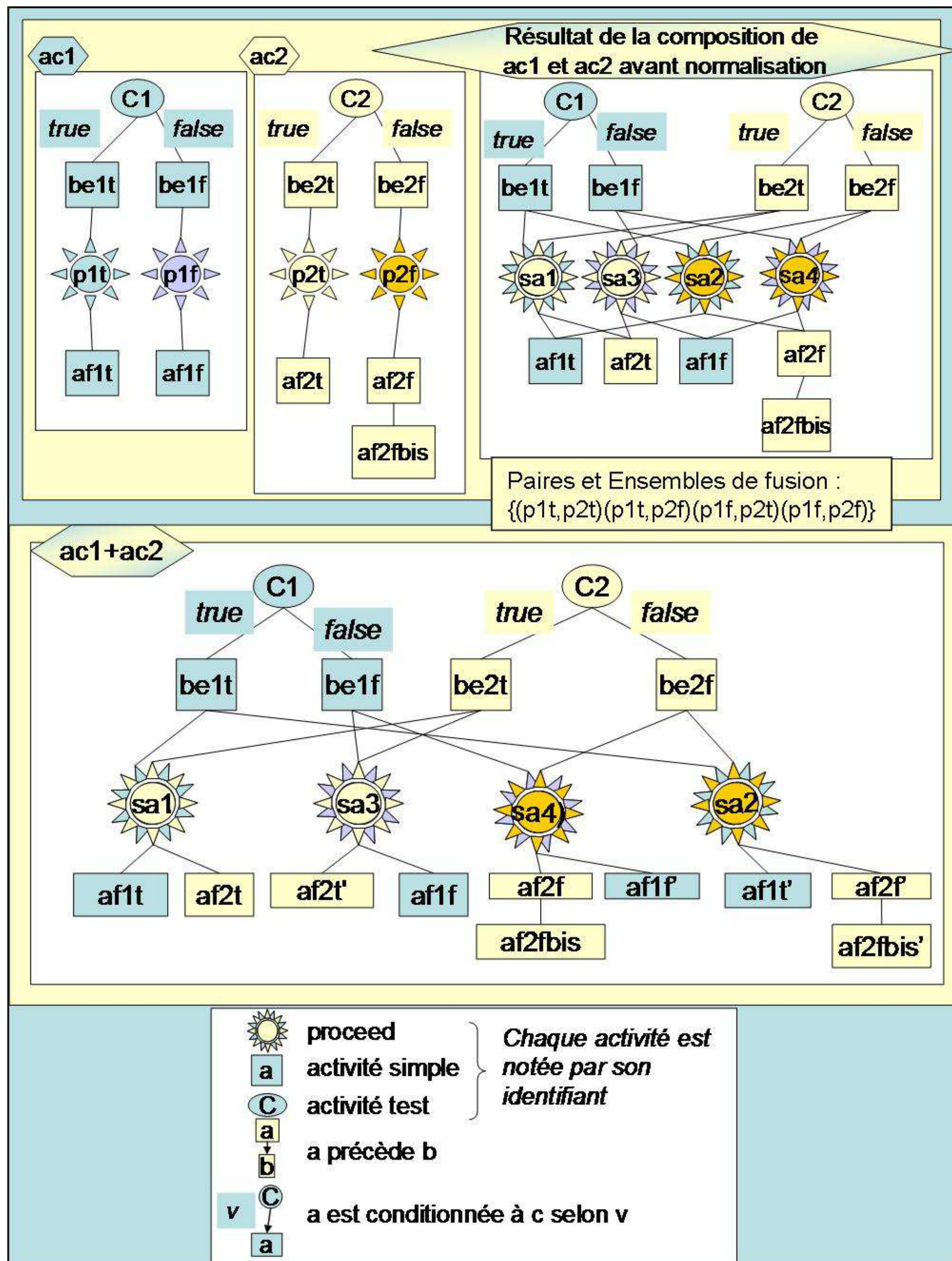
   A cette étape des **conflits** dits de "**fusion**" peuvent être détectés lorsque l'on ne parvient pas à calculer la fusion sur la base de l'ensemble des activités et relations en présence.
4. Application des transformations (cf. 3.3).
   La détermination de l'ordre d'application des transformations et l'application ordonnée des transformations va permettre d'assurer l'indépendance vis-à-vis de l'ordre des activités.

   La fonction $\sum_D$ dépendante domaine capture cette étape.

   A cette étape des **conflits** dits de "**transformation**" peuvent apparaître ; par exemple lorsque l'on ne parvient pas à calculer un ordre d'application des transformations qui assure un résultat unique au sens de l'équivalence.
5. Normalisation partielle (cf. 3.5)
   La normalisation partielle permet de rester dans le domaine des activités composites valides qui sert d'hypothèse de base à toutes les fonctions.

   La fonction $PartiallyNormalize_D$ dépendante domaine capture cette étape.

   A cette étape des **conflits** correspondant à "**impossible normalisation partielle**" peuvent apparaître lorsque l'on ne parvient pas à normaliser l'ensemble d'activités et de relations obtenu. Cela peut se produire, par exemple, si un circuit a été détecté que la fonction de normalisation partielle ne sait pas résoudre.
6. La fusion donnant lieu à la création de nouvelles activités, de nouveaux ensembles de fusion peuvent apparaître. Le procédé est donc réitéré tant qu'il existe des ensembles de fusion.
7. Le résultat final (quand il n'y a plus d'ensemble de fusion) est alors un ensemble normalisé d'activités simples liées par des relations de précédence et de conditions. Cet ensemble est alors validé et éventuellement transformé pour se conformer au domaine ciblé, dans ce cas nous utilisons la normalisation sur Domaine vue au chapitre précédent (cf. 2.3.5).




Cette figure présente un exemple de composition de deux activités composites définies dans le cadre du domaine fil rouge. Les deux activités composites *ac1* et *ac2* sont composées autour de leurs pivots respectifs (activités *proceed*) pour engendrer la nouvelle activité composite *ac1 + ac2*. La fusion des activités pivots engendrent de nouvelles activités (sa*). La représentation en haut à droite montre le résultat de la composition avant la normalisation.

FIGURE 3.1: EXEMPLE DE COMPOSITION D'ACTIVITÉS COMPOSITES DANS LE DOMAINE FIL ROUGE

## 3.2 Activités pivots et ensembles de fusion

Étant données des activités composites et donc des ensembles d'activités simples liées par des relations de précédence et de condition, la composition repose sur la détermination des activités simples à "fusionner". Ces activités simples potentiellement fusionnables sont dites activités **pivots**.
Dans l'application de composition des *interactions* (cf. §6), les activités pivots seront identifiées par des mots clefs dans l'information de l'action (par exemple *proceed*, *delegate*), tandis que dans la composition d'*orchestrations* (cf. §7), toute invocation de service sera potentiellement un pivot. Ainsi à l'instar de [FV07], la détermination des activités à fusionner est dépendante domaine et peut impliquer des interactions avec l'utilisateur du système de composition.
Nous faisons donc le choix de déterminer les activités pivots à fusionner par paire. Nous parlons alors de **paires de fusion**.
La détection des paires de fusion dépend des domaines cibles. Elle peut ainsi dépendre d'une quantification (par exemple une expression régulière dans le cadre d'une composition d'aspects), par la reconnaissance de "role" [FFR+07] ou par surimposition [ALB+07] comme dans la fusion d'orchestrations [NBFKR07] où c'est la présence d'activités "redondantes" qui permet de construire les paires de fusion.
Nous généralisons ici cette fonction, ce qui nous permet de construire l'algorithme de composition indépendamment du domaine cible, tout en explicitant les propriétés attendues de cette fonction.

**Fonction sur domaine 3** (FD3) : *Paires de fusion*
La fonction $detectMergePairs_D$ définie de $\mathcal{SA}^* \times \mathcal{R}_p \times \mathcal{R}_c \longrightarrow (\mathcal{SA} \times \mathcal{SA})^* \vee \Upsilon$ calcule pour un ensemble d'activités, une relation de précédence et une relation de condition donnés, les paires de fusion potentielles.
Une erreur dites **conflit de pivot** se produit lorsque des activités qui auraient dû être fusionnées ne peuvent pas l'être.

Nous distinguons donc les activités non fusionnables au sens où il n'existe pas entre elles de correspondance (nous ne cherchons pas à composer une affectation et une invocation de service par exemple), de celles qui devraient être composées mais qui ne respectent pas les contraintes autorisant leur composition. Ces contraintes sont propres au domaine (deux redéfinitions d'une même activité ne sont pas fusionnables dans le domaine des interactions [1]).
Sur la base de ces paires de fusion, nous calculons des ensembles de fusion comme des ensembles d'activités fusionnables.

**Fonction 5** (F5) : *Ensembles de fusion*
La fonction $detectMergeSets$ définie de $\mathcal{SA}^* \times \mathcal{R}_p \times \mathcal{R}_c \longrightarrow (\mathcal{SA}^*)^* \vee \Upsilon$ calcule pour un ensemble d'activités, une relation de précédence et une relation de condition données, les ensembles de fusion.
Cette fonction est définie sur la base de la précédente, en considérant les paires de fusion comme constituant un graphe [2] et en calculant les cliques maximales [3].

1. dites délégations
2. Graphe dont les nœuds sont les identifiants et les arcs la présence de paires les unissant.
3. une clique est un ensemble de sommets deux-à-deux adjacents.

Par exemple, si nous avons obtenu les paires de fusion :
$\{(a1, a2), (a1, a2'), (a2, a3), (a2', a4), (a1, a3)\}$
l'ensemble des ensembles de fusion sera : $\{(a1, a2, a3), (a1, a2'), (a2', a4)\}$

**Propriété 17 (Cardinalité des paires et des ensembles de fusion)** *Soit n le nombre d'activités pivots.*
*Le maximum de paires qui peut être détecté est de : n(n-1)/2 paires ; dans ce cas il y a un unique ensemble de fusion.*
*Dans le cas général, la cardinalité de l'ensemble des ensembles de fusion est bornée d'après Moon and Moser par* $3^{n/3}$ .

**Propriété 18 (Complexité du calcul des paires et des ensembles de fusion)** .
*Soit n le nombre d'activités pivots.*
*Le calcul des paires de fusion a une complexité en* $O(n^2)$.
*Il a été démontré dans [TTT06], qu'on peut construire un algorithme qui calcule toutes les cliques maximales dans le pire des cas en* $O(3^{n/3})$.

Dans les domaines ciblés, la complexité des calculs n'est pas toujours proportionnelle au nombre des activités pivots ou à la cardinalité des ensembles calculés.

**Propriété 19 (Paires de fusion et graphes triangulés)** .
*Lorsque la fonction* $detectMergePairs_D$ *produit des graphes triangulés* [4]*, le nombre d'ensembles de fusion est borné par le nombre de pivots.*

**Propriété 20 (Paires de fusion et indépendance de l'ordre)** .
*La fonction* $detectMergePairs_D$ *est indépendante de l'ordre de traitement des activités si les paires de fusion ne dépendent pas de l'ordre de traitement des activités.*

Cette propriété établit que le résultat ne doit pas dépendre de l'ordre de traitement des activités. Du fait que nous travaillons sur des ensembles d'activités simples cette propriété établit simplement qu'elle est déterministe. Dans [KFJ07], le choix des paires de fusion porte sur l'ensemble des aspects et est ordonné, quand c'est possible, de manière à former une suite ordonnée de points de jonction. Cette approche souligne l'importance de détecter à cette étape les conflits de pivots (dans ce cas là, lorsque il n'est pas possible d'ordonner les points de jonctions) et pose la question d'ordonner les ensembles de fusion. Même si l'algorithme proposé n'exclut pas cette possibilité, elle n'est pas directement étudiée dans ce travail.

4. Un graphe est triangulé (graph chordal) s'il ne contient pas un circuit de longueur quatre ou plus sans corde. Cette configuration se produit quand les paires sont associatives.

`Exemple Fil rouge:` **Activités pivots et ensembles de fusion**

**Algorithme :**
Dans ce domaine, les seules activités pivots sont celles qui font référence à une information *proceed*.

$detectMergePairs_{FilRouge}$ : La détection des paires de fusion consiste à déterminer les activités pivots (*proceed*) qui ne s'excluent pas. Étant données deux activités pivots non exclusives, elles doivent alors former une paire. Cependant si le nombre des variables en entrée est différent alors la paire ne peut pas être construite et dans ce cas, il y a un conflit de pivot. Les activités composites ne peuvent être composées que si l'arité des activités pivots est la même.

**Exemple :**
Dans la figure 2.7, les paires de fusion construites entre `ca1, ca2` et `ca3` sont :
`(a13,a22) (a13,a32) (a13,a36) (a22,a32)(a22,a36) (a14,a22) (a14,a32) (a14,a36)`
Et les ensembles de fusion correspondant sont :
`(a13,a22,a32) (a13,a22,a36) (a14,a22,a32) (a14,a22,a36)`

**Propriétés :**

**Propriété DFR 1 (éléments d'un ensemble de fusion)** *Les éléments des ensembles de fusion correspondent à des activités issues d'activités composites toutes différentes.*

**Preuve.** En effet, pour que deux activités pivots ne s'excluent pas, elles doivent appartenir à des activités composites différentes (d'après la contrainte 1 qui est posée sur le domaine, page 41).

**Propriété DFR 2 (exclusion des ensembles de fusion)** *L'union des gardes portant sur les activités d'un ensemble de fusion est exclusive vis-à-vis de tout autre ensemble de fusion.*

**Preuve.** En effet, soient deux ensembles de fusion $e1$ et $e2$, soit ils n'ont pas d'activités en commun et dans ce cas il existe $a1 \in e1$ et $a2 \in e2$ telles que ces activités s'excluent sinon elles auraient formées des paires, soit il existe trois activités $a1, a2, a3$ telle que $\{a1, a2\} \subseteq e1$ et $\{a1, a3\} \subseteq e2$ et il n'existe pas de paire de fusion entre $a2$ et $a3$ (sinon les trois activités seraient dans la même clique) ; or la seule possibilité pour que deux activités pivots ne constituent pas une paire est qu'elles s'excluent. En conséquence l'union des conditions de $e1$ et de $e2$ s'excluent.

**Propriété DFR 3 (cardinalité des ensembles de fusion)** *Soient $ca_1...ca_n, n$ activités composites composées chacune de $p_i$ pivots. Le nombre maximum de paires de fusion sera de : $\sum_{i=1}^{i=n-1} \sum_{j=i+1}^{j=n} p_i * p_j$.*
*Le nombre de cliques maximales sera lui borné par : $p_1 * p_2 * ... * p_n$*
*La cardinalité maximale d'un ensemble de fusion est de n, d'après la propriété 1 du domaine Fil Rouge.*

La complexité du calcul des paires est donc réduite par le fait que nous ne comparons pas toutes les activités pivots deux à deux mais seulement les activités pivots appartenant à des activités composites différentes. De plus dans ce domaine le nombre des activités pivots est restreint puisqu'il est proportionnel aux nombres de branches dans une activité composite.

**Propriété DFR 4 (indépendance de l'ordre)** *Le calcul des paires de fusion est indépendant de l'ordre de traitement des activités puisque ce calcul opère sur l'ensemble des activités et calcule toutes les paires. En cas de conflit de pivot, le résultat est une erreur indépendante des calculs antérieurs. La propriété 20 est donc vérifiée sur le domaine Fil Rouge.*

## 3.3 Transformations et compositions des transformations

A la base de la composition des activités sont définies des transformations qui permettent de modifier les activités simples et les relations entre elles.
Nous notons $\mathcal{F}$ l'espace de ces transformations.
Bien qu'il puisse partiellement être construit sur l'espace minimal des fonctions atomiques de constructions de modèles telles que définies par Gregory De Fombelle dans [dF07], nous le considérons comme dépendant domaine et le détaillons dans les applications.
Nous notons $TodoR_{\mathcal{F}}$ l'ensemble des transformations dans $\mathcal{F}$ définies sur :
$\mathcal{SA}^* \times \mathcal{R}_p \times \mathcal{R}_c \longrightarrow \mathcal{SA}^* \times \mathcal{R}_p \times \mathcal{R}_c$.

*Equivalence de transformations*

Deux transformations $todo_1$ et $todo_2$ sont équivalentes si :
$\forall(l^0, R_p^0, R_c^0)\ todo_1(l^0, R_p^0, R_c^0) = (l^1, R_p^1, R_c^1)$ , $todo_2(l^0, R_p^0, R_c^0) = (l^2, R_p^2, R_c^2)$ et
$\exists\sigma, l^1 \equiv_\sigma l^2, R_p^1 \equiv_\sigma R_p^2, R_c^1 \equiv_\sigma R_c^2$

**Quelques transformations** Sur la base du modèle des activités, certaines transformations sont récurrentes. Nous les présentons ici. Elles seront complétées en fonction des domaines.

1. $remove_{(activity)}$ : retire l'activité de l'ensemble des activités.
   $remove_{(OldAs)}(AL, R_p, R_c) = (AL', R_p, R_c)$ où $AL' = AL \backslash \{OldAs\}$
   $remove_{(id_{activity})}$ : retire l'activité de la liste des activités.
   $remove_{(id_{OldAs})}(AL, R_p, R_c) = (AL', R_p, R_c)$ où $AL' = AL \backslash \{getSa_S(id_{OldAs})\}$
   Le retrait d'une activité inexistante ne déclenche pas d'erreur.

2. $substituteIn_{(oldVar,newVar)}$ : substitue dans les activités l'ancienne variable par la nouvelle.
   $substituteIn_{(OldVar,NewVar)}(AL, R_p, R_c) = (AL', R_p, R_c)$ où $AL' = \{a \in AS | \exists a_i \in AL, \sigma(a_i) = a\}$ avec $\sigma = \{(OldVar, NewVar)\}$

3. $substituteActivity_{(oldId,newId)}$ : remplace dans les relations de précédence et de condition les occurrences de $oldId$ par $newId$.
   $substituteActivity_{(oldId,newId)}(AL, R_p, R_c) = (AL, R'_p, R'_c)$ où
   $R_{p_{oldId}} = \{(oldId, X) \in R_p\} \cup \{(X, oldId) \in R_p\}$
   $R_{p_{oldId}} \cup \overline{R_{p_{oldId}}} = R_p$,
   $R'_p = \{(x, newId) | (x, oldId) \in R_{p_{oldId}}\} \cup$
   $\quad \{(newId, y) | (oldId, y) \in R_{p_{oldId}}\} \cup \overline{R_{p_{oldId}}}$
   $R_{c_{oldId}} = \{(C, oldId, V) \in R_c\} \cup \{(oldId, I, V') \in R_c\}$
   $R_{c_{oldId}} \cup \overline{R_{c_{oldId}}} = R_c$,
   $R'_c = \{(C, newId, V) | (C, oldId, V) \in R_{c_{oldId}}\} \cup$
   $\quad \{(newId, I, V') | (oldId, I, V') \in R_{c_{oldId}}\} \cup \overline{R_{p_{oldId}}}$

**Exemple Fil rouge: Transformations**

Afin de composer les activités composites dans ce domaine, nous avons besoin des transformations *remove* et *substituteIn* telles que définies précédemment.

La composition de transformations $\{todo_1, ..., todo_n\} \in TodoR_{\mathcal{F}}^n$ dépend des domaines cibles. Nous définissons à présent cette fonction.

**Fonction sur domaine 4** (FD4) : *Composition de transformations*

La fonction $\sum_D$ définies de $TodoR_{\mathcal{F}}^n$ dans $TodoR_{\mathcal{F}} \vee \Upsilon$ calcule la transformation résultante de la composition de l'ensemble des transformations.

Nous notons : $\sum_{D i=1}^{i=n} todo_i(l^0, R_p^0, R_c^0)$ l'évaluation d'un ensemble de transformations sur $(l^0, R_p^0, R_c^0)$. Cette composition des transformations est caractérisée en fonction des domaines cibles.

Lorsque la transformation résultante ne peut pas être calculée, nous considérons qu'il y a un **conflit de transformation**.

**Propriété sur Domaine 1 ( Composition des transformations indépendante de l'ordre )**
*La composition des transformations est indépendante de l'ordre si quelque soit l'ordre des transformations, la transformation résultante est équivalente, c-à-d. :*
*étant donné un ensemble de transformations* $E = \{todo_i | todo_i \in TodoR_{\mathcal{F}}\}$,
$\forall (l^0, R_p^0, R_c^0), \exists (l, R_p, R_c)$ $\sum_{D i=1}^{i=n} todo_i(l^0, R_p^0, R_c^0) \equiv_\sigma (l, R_p, R_c)$ *quelque soit l'ordre des éléments dans **E**.*

Cette propriété n'exprime pas une indépendance dans l'ordre d'évaluation, mais l'existence d'un ordre d'évaluation assuré par la fonction $\sum_D$ qui garantit l'équivalence des résultats.

Relativement aux quelques transformations définies précédemment, nous pouvons noter que :
- La composition des substitutions d'activités n'est possible que si une activité ne doit jamais être remplacée par des activités différentes ; dans le cas contraire, une erreur est levée.
- Si nous composons des substitutions selon la fonction définie au paragraphe 2.3.7, l'ordre a de l'importance.
- Les transformations relatives aux retraits d'activités ou substitutions de variables ne sont pas en interaction avec la substitution d'activités ; elles n'interviennent pas sur les mêmes éléments.
- Il est préférable d'appliquer le retrait d'activités avant la substitution de variables pour éviter des substitutions inutiles de variables.
- La substitution d'activités peut créer des circuits lors de son application. Il n'est pas possible de les identifier à la composition des transformations ; les circuits dépendent du jeu de données.

**Exemple Fil rouge: Composition des transformations**

**Algorithme :**
Nous définissons la composition des transformations $\sum_{FilRouge}$ comme l'application de l'ensemble des retraits d'activités puis l'application des substitutions composées comme suit. L'application des retraits en premier nous permet d'éviter des substitutions inutiles.

*Composition des substitutions de variables* Les substitutions $\sigma_1 ... \sigma_n$ sont composées en recherchant l'unificateur principal, i.e. en considérant que chaque substitution $\sigma_i = \{(v_i^1, r_i^1), ...\}$ est un problème de la forme $s_i = \{v_i^1 = r_i^1, ...\}$ où les $v_i$ et $r_i$ sont des variables et les $r_i$ n'apparaissent jamais en partie gauche de l'affectation, et en recherchant l'unificateur principal $\sigma$ du problème $\bigcup_{i=1}^{i=n} s_i$.
Par exemple : $x = v1, y = v1, x = v3, y = v4$ aura pour unificateur principal : $\{(x, v5), (y, v5), (v1, v5), (v3, v5), (v4, v5)\}$.
Comme nous ne manipulons que des variables, il n'y a pas d'échec possible.

**Propriété DFR 5** *Composition des transformations indépendante de l'ordre à l'équivalence près.*

**Preuve.** L'application des activités de retraits donne un résultat identique quelque soit l'ordre des retraits. La composition des substitutions repose sur la construction d'un unificateur. Les propriétés suivantes s'appliquent donc. (i) Il y a unicité de unificateur principal au renommage près des variables. Or la relation d'équivalence prend en compte cette caractéristique. (ii) Nous sommes certain de trouver un unificateur car les cas d'échec de l'algorithme d'unification ne s'applique pas : (i) nous ne manipulons pas des termes mais uniquement des variables (pas d'utilisation de la règle de décomposition et donc de conflit), (ii) les variables en partie droite n'apparaissent jamais en partie gauche dans les substitutions élémentaires (test de cyclicité inutile).

La richesse des transformations supportées et leur capacité à se composer sont des éléments clefs de l'algorithme de composition. Les transformations sont les éléments d'abstraction autorisant une expression de haut niveau des compositions [BBF+06]. Ces transformations pourront elles-même dépendre d'informations additionnelles données par le développeur comme l'expression des correspondances entre les variables. En définissant formellement les transformations au sens où leur action est connue, nous pouvons établir les bases d'un algorithme de composition dont nous maîtrisons les propriétés.

## 3.4 Fusion d'activités simples au sein d'un ensemble d'activités

A partir d'un ensemble d'activités fusionnables relativement à un ensemble donné d'activités liées par une relation de précédence et de condition, il s'agit de déterminer les nouvelles activités résultantes de la fusion, l'ensemble des nouveaux éléments de précédence et de conditions et l'ensemble des transformations à opérer sur l'ensemble des activités résultantes [5].

5. De telles fonctions sont considérées comme des transformations d'ordre supérieur selon [Pas06b], car elles prennent en entrée entre autre un ensemble de transformations et retourne une nouvelle transformation ; c'est une transformation de transformations.

**Fonction sur domaine 5** (FD5) : *Fusion d'activités simples*

$mergeSimpleActivities_D$ :

$\mathcal{SA}^+ \times \mathcal{R}_p \times \mathcal{R}_c \times \mathcal{SA}^n, n > 1, \longrightarrow (\mathcal{SA}^* \times \mathcal{R}_p \times \mathcal{R}_c \times TodoR^*_{\mathcal{F}}) \cup \Upsilon,$

Lorsque la fusion ne crée pas d'erreur, elle est définie comme suit :

$mergeSimpleActivities_D(Activities, R_p, R_c, MergeSet)$
$= (NewActivities, NewR_p, NewR_c, todoR)$ où

- $NewActivities$ représente les activities qui ont été créées, cet ensemble peut être vide
  $NewActivities \cap Activities = \emptyset$
- $NewR_p$ représente les couples de précédence ajoutés,
  un des paramètres de ces couples est nécessairement une nouvelle activité
  $\Rightarrow NewR_p \cap R_p = \emptyset$,
- $NewR_c$ représente les triplets de condition ajoutés,
  un des paramètres de ces relations est nécessairement une nouvelle activité
  $\Rightarrow newR_c \cap R_c = \emptyset$,
- $todoR$ correspond à l'ensemble des transformations à appliquer sur le résultat global de la fusion.

Lorsque la fusion des activités simples engendre une erreur, il y a **conflit de fusion** entre les activités.

`Exemple Fil rouge:` **Fusion d'activités pivots**

**Algorithme :**

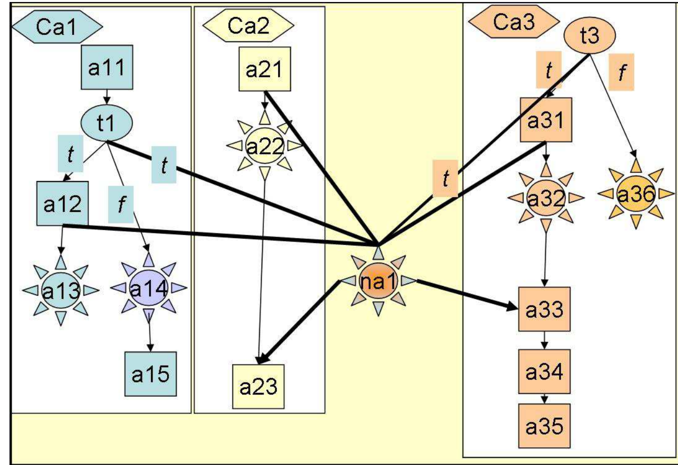
La fusion des activités dans le domaine Fil Rouge est définie comme suit.
$\forall LA \in \mathcal{SA}, R_p \in \mathcal{R}_p, R_c \in \mathcal{R}_c, \{s_1..s_n\} \subseteq LA$ où $s_1..s_n$, n activités pivots d'identifiant respectif $id_1..id_n$ et correspondant à un ensemble de fusion,
$mergeSimpleActivities_{FilRouge}(LA, R_p, R_c, \{s_1..s_n\}) =$
$(\{a\}, newR_p, newR_c, Sub \cup \{remove_{(id_1)}...remove_{(id_n)}\})$ où les différentes composantes sont obtenues comme suit :

- *$Sub$ : Substitutions de variables*
  Soient $\{i1_1...i1_k\}...\{in_1...in_k\}$ les variables d'entrées respectives de $s_1..s_n$,
  on crée de nouvelles variables $\{i_1...i_k\}$
  et pour chaque variable en entrée, on génère les composants de substitutions :
  $Sub = \{substituteIn(im_i, i_i) | im_i \in inputs(s_m), m \in 1..n, i \in 1..k\}$
- *$a$ : création d'une nouvelle activité simple* d'identifiant $id$, dont l'action a pour information *proceed* et les variables d'entrées sont $\{i_1...i_k\}$ et les variables de sorties $\{result\}$.
- *$newR_p$ : création de nouveaux éléments de précédence* :
  $NewR_p = \{pred(x, id) | pred(x, id_i) \in R_p, i \in 1..n\} \cup \{pred(id, x) | pred(id_i, x) \in R_p, i \in 1..n\}$
- *$newR_c$ : création de nouveaux éléments de condition :*
  $NewR_c = \{cond(c, id, v) | cond(c, id_i, v) \in R_c, i \in 1..n\}$

**Exemple :**

La fusion des ensembles d'activités présentées dans la figure 3.2 donne lieu à la création de 4 activités *proceed* comme suit : `(a13,a22,a32) => na1`; `(a13,a22,a36)=>na2`; `(a14,a22,a32)=>na3`; `(a14,a22,a36)=>na4`.
La figure 3.2 présente l'activité *na1* créée et les éléments de précédence et de condition associés qui ont été créés. Nous ne présentons pas graphiquement les relations créées par la fusion pour les autres activités pour des raisons de clarté.

**Propriété 21 (Pas de modification des activités par fusion)** *La fusion d'activités simples au sein d'une liste d'activités ne modifie ni l'ensemble d'activités, ni les relations de précé-*




Cette représentation graphique fait référence à l'exemple de la page 36. La fusion des activités `a13`, `a22` et `a32` correspondant respectivement aux actions `proceed(x)`, `proceed(y)`,`proceed(z)` donne lieu à :
- la création de l'activité `na1` correspondant à l'action `proceed(w1)`
- les couples de précédence : `{(a12,na1), (a21,na1), (a31,na1), (na1,a23), (na1,a33)}`,
- les conditions `{(t1,na1,true),(t3,na1,true)}`
- et les transformations : `{substituteIn(x,w1),substituteIn(y,w1),substituteIn(z,w1), remove(a13),remove(a22),remove(a32)}`

FIGURE 3.2: EXEMPLE DE FUSION D'ACTIVITÉS SIMPLES (A13,A22,A32)

*dence et de conditions.*

Les transformations *todo* sont appliquées à l'étape suivante à l'ensemble des activités composites et aux nouvelles activités simples et éléments de relations créés.
Ne pas appliquer les transformations sans les composer auparavant permet de prendre en compte les cas où une fusion d'activités simples conduit à retirer des activités elles-même fusionnables. Ce problème a également été évoqué dans les travaux sur la composition de scénarios [KFJ07].

**Propriété sur Domaine 2 (Fusion indépendante de l'ordre des activités)** *La fonction $mergeSimpleActivities_D$ est indépendante de l'ordre des éléments dans les ensembles de fusion lorsque :*
*Soit $S1 \equiv_\sigma S2$ , 2 ensembles de fusion équivalents,*
$mergeSimpleActivities_D(LA, R_p, R_c, S1) \equiv_\sigma mergeSimpleActivities_D(LA, R_p, R_c, S2)$

*c-à-d.*

$$mergeSimpleActivities_D(LA, R_p, R_c, S1) = (NewActs12, newR12_p, newR12_c, todoR12)$$
$$mergeSimpleActivities_D(LA, R_p, R_c, S2) = (NewActs21, newR21_p, newR21_c, todoR21)$$

*et* $\exists\sigma | NewActs12 \equiv_\sigma NewActs21, newR12_p \equiv_\sigma newR21_p, newR12_c \equiv_\sigma newR21_c, todoR12 \equiv_\sigma todoR21$

**Propriété sur Domaine 3 (Pas de création de circuit )** *La fusion d'activités pivots ne crée pas de circuit si :*
$\forall AL \in \mathcal{SA}^+, \forall R_p \in \mathcal{R}_p, \forall R_c \in \mathcal{R}_c, \forall s \subseteq AL$*, $s$ étant un ensemble de fusion :*

$mergeSimpleActivities_D(AL, R_p, R_c, s) = (NewActivities, newR_p, newR_c, \{todo_1..todo_n\})$
$\Longrightarrow$

$(AL^+, R_p^+, R_c^+) = \sum_{D\,i=1}^{i=n} todo_i(AL \cup NewActivities, R_p \cup newR_p, R_c \cup newR_c)$
$\forall a_i \in AL+$
$pred^*(R_p^+{}_{|AL^+}, a_i) \bigcap succ^*(R_p^+{}_{|AL^+}, a_i) = \emptyset \wedge$
$cond^*(R_c^+{}_{|AL^+}, a_i) \bigcap succ^*(R_p^+{}_{|AL^+}, a_i) = \emptyset$

Cette propriété vérifie que l'application des transformations issues de la fusion simple ne créeront pas de cycles. Il s'agit donc d'une vérification "locale" au sens où nous ne disposons au niveau de cette fonction que des informations relatives à un seul ensemble de fusion.

**Propriété sur Domaine 4 (Pas d'incohérence)** *La fusion d'activités pivots ne crée pas d'incohérence si :*
$\forall AL \in \mathcal{SA}^+, \forall R_p \in \mathcal{R}_p, \forall R_p \in \mathcal{R}_c, \forall s \subseteq AL$
$mergeSimpleActivities_D(AL, R_p, R_c, s) = (NewActivities, newR_p, newR_c, \{todo_1..todo_n\})$
$\Longrightarrow$

$(AL^+, R_p^+, R_c^+) = \sum_{D\,i=1}^{i=n} todo_i(AL \cup NewActivities, R_p \cup newR_p, R_c \cup newR_c)$
$\forall a_i \in AL^+, condTrue^*(R_c^+{}_{|AL^+}, a_i) \bigcap condFalse^*(R_c^+{}_{|AL^+}, a_i) = \emptyset$

**`Exemple Fil rouge:` Propriétés de la fusion**

**Propriété DFR 6 (Pas de création de circuit)** *La propriété 3 est vérifiée sur le domaine Fil Rouge.*
**Preuve.** Si *a1... an* sont les activités à fusionner, elles appartiennent initialement à des activités composites différentes à cause de la propriété 2 du domaine Fil Rouge donc il n'y a pas d'intersection entre les ensembles de leurs prédécesseurs et successeurs.

**Propriété DFR 7 (Cohérence des activités fusionnées)** *La propriété 4 est vérifiée sur le domaine Fil Rouge.*

**Preuve.** Seules des activités non exclusives sont fusionnées. L'activité résultat de la fusion étant conditionnée par l'union des conditions sur les activités fusionnées, elle n'est pas assujettie à des conditions qui s'excluent.

**Propriété DFR 8 (Indépendance vis-à-vis de l'ordre des activités)** *La propriété 2 est vérifiée sur le domaine Fil Rouge.*

**Preuve.** Nous avons montré que la composition des transformations est indépendante de l'ordre par la propriété 5 du domaine Fil Rouge et la détermination des nouvelles activités et relations ne dépend pas de l'ordre des activités dans les ensembles.

## 3.5 Normalisation partielle

A partir des ensembles d'activités et les relations résultantes des transformations, il peut être nécessaire d'opérer une opération de normalisation dite partielle qui vérifie que le graphe d'activités résultant est valide et éventuellement le corrige.

**Fonction sur domaine 6** (FD6) : *Normalisation Partielle*
La fonction de normalisation partielle est définie de :
$\mathcal{SA}^+ \times \mathcal{R}_p \times \mathcal{R}_c \longrightarrow (\mathcal{SA}^+ \times \mathcal{R}_p \times \mathcal{R}_c) \vee \Upsilon.$

soit $(la, R_p, R_c), PartiallyNormalize_D(la, R_p, R_c) = (la', R'_p, R'_c)$ si la normalisation partielle sur domaine est possible.
Lorsque la normalisation ne peut pas être opérée une erreur **impossible normalisation partielle** est levée.

**`Exemple Fil rouge`: Normalisation partielle**

La fonction de normalisation partielle dans le domaine Fil rouge fait appel à la fonction 2.3.9 de normalisation que nous avons définie à la page 44. La validité du graphe d'activités est réparée par cette fonction.
Elle ne lève pas d'erreur.

## 3.6 Composition d'un ensemble d'activités composites

La composition d'activités composites commence par copier les activités composites et renommer les variables qu'elles comportent de manière unique afin d'éviter les interférences entre les variables. Ensuite la composition procède par circuit : (1) détection des ensembles de fusion dans la liste d'activités simples, (2) fusion des activités simples, (3) construction de la nouvelle liste d'activités, (4) normalisation puis nouvelle itération s'il existe encore des paires de fusion. Enfin le résultat de la fusion est validé au niveau domaine applicatif avant d'être considéré comme une nouvelle activité composite.
La détermination des ensembles de fusion de manière cyclique vient de ce que (i) lors d'interactions avec l'utilisateur celui-ci peut remettre à une autre itération la fusion d'un ensemble d'activités[6], (ii) de nouveaux ensembles de fusion peuvent apparaître suite à certaines fusions. La normalisation partielle à chaque tour de boucle nous permet de travailler sur des activités valides.
La normalisation sur domaine assure que les activités construites respectent bien les contraintes imposées par le domaine applicatif. Nous ne vérifions pas cet aspect à chaque tour de boucle parce que le mécanisme de composition a justement comme ambition de conduire à une activité composite valide dans un domaine et qu'il est donc possible que l'ensemble des activités construit ne respecte pas les contraintes du domaine tant que tous les ensembles de fusion n'ont pas été résolus. Par contre nous opérons des normalisations partielles qui assurent que l' ensemble d'activités est toujours valide, propriété requise par les fonctions sur domaine.
Si la composition d'activités composites ne génère pas d'erreur, nous disons que la **composition est valide**.

**Fonction 6** (F6) : *Composition d'un ensemble d'activités composites*

La composition d'un ensemble d'activités composites est notée $\bigoplus_D$[7].
Elle est définie de $\mathcal{CA}^+ \longrightarrow \mathcal{CA} \cup \Upsilon$
Les activités composites initiales sont valides.
Soit $Set = \{ca_1, ..., ca_n\}$
$\bigoplus_D Set = \bigoplus_D \{ca_1, ..., ca_n\} = \bigoplus_{D i=1}^{i=n} ca_i$

6. Cela se produit par exemple lorsque la fusion de certains ensembles d'activités est difficile à déterminer alors que la résolution d'autres ensembles peut réduire les possibilités des fusions (par exemple par unification d'invocations de services, des unifications de variables peuvent être automatiquement propagées à l'ensemble des activités). Nous étudierons cet aspect dans la composition des orchestrations (cf. §7)

7. Nous faisons apparaître le domaine comme discriminant pour cette fonction, même si la dépendance vis-à-vis du domaine n'est pas directe mais associée aux fonctions utilisées.

- $\bigoplus_{D i=1}^{i=n} ca_i$ échoue si
  - la détection des ensembles de fusion lève une erreur de conflit de pivot,
  - ou si la fusion d'un ensemble d'activités pivot échoue,
  - ou si les normalisations partielles ou globales en fonction du domaine échoue.
- Dans le cas où il n'y a pas d'échec :
  $\bigoplus_{D i=1}^{i=n} ca_i = ca_r$ où $ca_r = (AL_r, R_{p_r}, R_{c_r})$ est une nouvelle activité composite construite selon l'algorithme suivant :
  Soient $i \in [1..n], ca'_i = copy(ca_i) = (id_i, AL_i, R_{p_i}, R_{c_i})$
  Soient $AL = \bigcup_{i=1}^{i=n} AL_i, R_p = \bigcup_{i=1}^{i=n} R_{p_i}, R_c = \bigcup_{i=1}^{i=n} R_{c_i}$
  1. Construire l'ensemble des ensembles de fusion :
     $detectMergeSets(AL, R_p, R_c) = \{s_1, ...s_n\}$
  2. Si l'ensemble des ensembles de fusion est vide,
     $ca = createCa(normalizeOnDomain_D(AL, R_p, R_c))$
     sinon,
     (a) pour chaque ensemble de fusion $s_i$, calculer :
     $mergeSimpleActivities_D(AL, R_p, R_c, s_i) = (NewActivities_i, R_{p_i}, R_{c_i}, todo_i)$,
     $todo_i = \{todo_1^i..todo_{max_i}^i\}$
     (b) Appliquer l'ensemble des transformations, sur l'union des ensembles issus des fusions simples :
     $AL' = AL \cup (\cup_{i=1}^{i=n} NewActivities_i)$
     $R'_p = (\cup_{i=1}^{i=n} R_{p_i} \cup R_p)$
     $R'_c = (\cup_{i=1}^{i=n} R_{c_i} \cup R_c)$
     $todo' = \cup_{i=1}^{i=n} todo_i = \{todo_1^1..todo_{max_n}^n\} = \{todoe_1..todoe_{max}\}$
     $(AL^+, R_p^+, R_c^+) = \sum_{D j=1}^{j=max} todoe_j(AL', R'_p, R'_c)$
     (c) Normaliser la nouvelle liste d'activités après restriction des relations de condition et de précédence :
     $partiallyNormalize_D(AL+, R_p^+{}_{|AL^+}, R_c^+{}_{|AL^+}) = (AL, R_p, R_c)$
     (d) reprendre au point 1.

La mise en œuvre de cet algorithme exige que les propriétés suivantes soient vérifiées.

**Propriété exigée 1 (Décroissance du nombre d'ensembles de fusion)** *Pour que cet algorithme termine, il est nécessaire que le nombre d'ensembles de fusion décroisse à chaque tour de boucle. Il convient donc de démontrer ce point dans les interprétations sur domaines.*

Notons que par construction de l'algorithme de composition, une activité composite résultante d'une composition d'activités, avant la normalisation sur domaine, ne peut pas contenir de paires de fusion.

**Propriété exigée 2 (Validité des activités ou Clôture)** *Lorsque les fonctions sur domaine exigent la validité des ensembles d'activités (absence de circuit, cohérence, ...), il convient de démontrer que les ensembles d'activités et les relations obtenues par fusion respectent ces propriétés ou de définir une fonction de normalisation partielle qui gère le problème.*

Notons que le respect des propriétés sur domaine 3 et 4 n'assure que partiellement la validité des activités avant la normalisation. En effet, l'application de l'ensemble des transformations peut introduire des circuits ou des incohérences.

**`Exemple Fil rouge:` Composition d'activités composites**

**Algorithme :**
L'algorithme s'applique sur la base des fonctions précédemment définies.

**Propriétés :**

Terminaison : L'algorithme termine parce que le résultat de l'application des étapes 1 et 2 suffit à ce qu'il n'y ait plus d'ensemble de fusion. En effet, la fusion des activités génère uniquement de nouvelles activités pivots qui s'excluent et qui en conséquence ne formeront pas des paires. Les activités pivots générées s'excluent d'une part parce que l'union des gardes des ensembles de fusion s'excluent (d'après la propriété 2 du domaine Fil Rouge) et l'activité résultante est assujettie à l'ensemble des gardes portant sur les activités dont elle est issue de part la fonction de fusion.

Acyclisme : L'absence de circuit a été démontrée par la propriété 6 du domaine Fil Rouge pour chaque fusion. Les transformations ne modifient pas la relation d'ordre. La normalisation ne crée pas de circuit d'après la propriété 15, page 45.

Cohérence : Les tests sur les activités ne s'excluent pas parce que les seuls éléments de la relation mis en jeu par la composition, portent sur les activités ajoutées dont nous avons démontré par la propriété 7 du domaine Fil Rouge qu'ils ne s'excluent pas. Les incohérences provoquées par les relations de précédence sont réparées par la normalisation partielle.

La composition obtenue après normalisation partielle assure donc que l'activité composite résultat est valide. Mais appartient-elle au domaine Fil Rouge ?

Présence des activités *proceed* : les nouvelles activités créés s'excluent, et toutes les activités pivots ont été remplacées par une activité équivalente sur laquelle porte toutes les conditions des activités fusionnées ;

Variables en entrée d'un *proceed* : la fusion n'a porté que sur des variables des activités pivots ; en conséquence, les nouvelles activités *proceed* ne portent que sur des variables et la variable de sortie est toujours *result* ; toutes les activités *proceed* ont en entrée les mêmes variables par unification ;

Déterminisme : l'union des relations de précédence et de conditions sur la nouvelle activité corrélée à l'unification des variables peut briser le déterminisme de l'ensemble d'activités résultants. En conséquence, la fonction de normalisation sur domaine peut échouer si nous ne l'étendons pas en autorisant ou en automatisant l'ajout de précédence en cas de détection de non déterminisme ;

Accès en lecture uniquement sur des variables valuées : la composition ne retire pas de précédence, en conséquence, si les activités composites respectaient la propriété, le nouvel ensemble d'activités respecte également cette propriété.

**Exemple Fil rouge: Composition d'activités composites : exemple**

**Exemple :**
La composition des activités composites *(ca1,ca2,ca3)* définies à la page 36 donne le résultat visualisé par la figure 3.4. Nous n'avons pas renommé les activités pour simplifier la compréhension, néanmoins, par composition, les activités sont copiées avant d'opérer les fusions et transformations. Le résultat avant normalisation est visualisé par la figure 3.3.
Nous ne présentons pas ici le code résultat car il ne peut pas être linéarisé sous la forme de séquences et de flots en introduisant les conditionnelles. Nous excluons donc l'expression des conditions et présentons ici le flot d'exécution correspondant à la validation des conditions sur la taille et la couleur *(`t1` et `t3` valuées à vrai)*.

```
((size = w3.size();(size>100); w3.reduce())
    // log('before') // (w3.isInColor(); cb := ColorPrinter.capacity()) );
result : = proceed(w3);
(log('after')//
   (ca := ColorPrinter.capacity(); c=ca-cb; Controler.notify('consumed',c)))
```

Nous constatons que (i) les relations de précédence ont été préservées, de même que les conditions, (ii) tandis que les activités ne correspondant pas à des pivots ne sont pas assujetties à des précédences autres que celles initiales. En conséquence, nous constatons que plusieurs des activités sont à présent "en parallèle", ce qui signifie qu'il n'y a pas de relation d'ordre en elles. Ainsi les accès en lecture à la variable *w3* par l'invocation de *reduce* et de *isInColor* sont réalisés dans un ordre non déterminé, ce qui n'est pas un problème tant que le sens de la lecture est clairement identifié. Si c'est le cas par l'hypothèse sur les prédicats, nous savons que dans le domaine de la programmation par objets, il s'agit d'une approximation. De fait la même variable aurait pu apparaître en sortie de l'invocation. Dans ce cas, nous considérons que nous ne sommes plus dans le domaine fil rouge, et la fonction de normalisation lève une erreur. Une autre approche aurait pu consister à définir la normalisation sur Domaine pour que l'on puisse ajouter une relation d'ordre entre les deux activités qui préserve le déterminisme de l'activité composite.

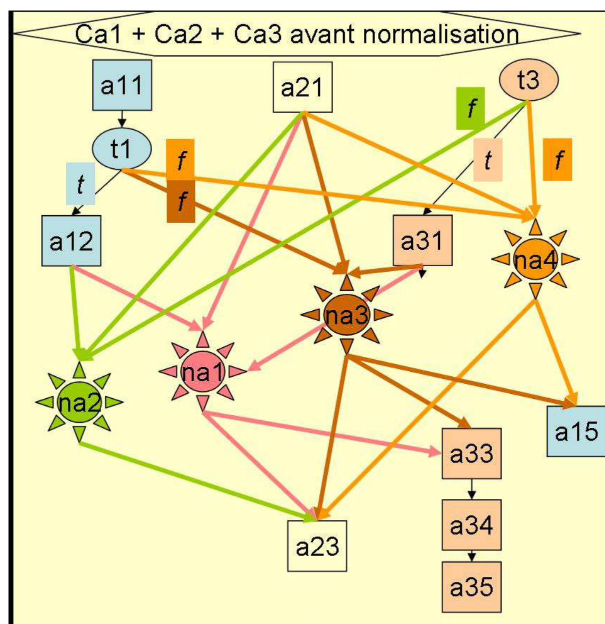



FIGURE 3.3: RÉSULTAT DE LA COMPOSITION AVANT NORMALISATION PARTIELLE

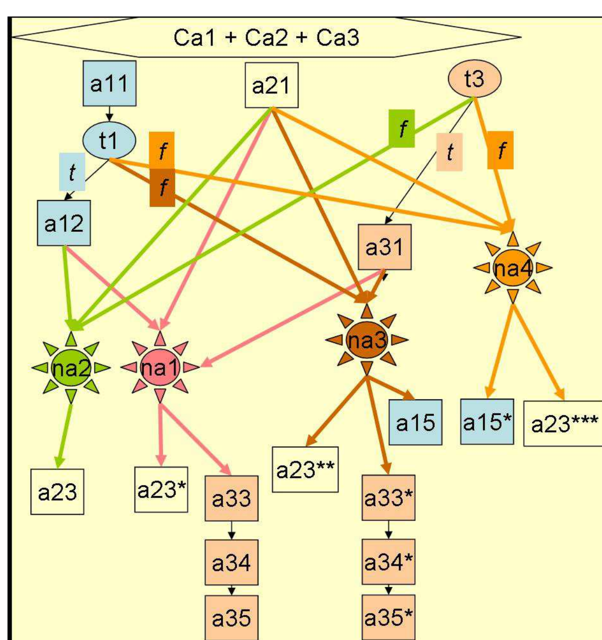



FIGURE 3.4: RÉSULTAT DE LA COMPOSITION APRÈS NORMALISATION PARTIELLE

# CHAPITRE 4

# Propriétés des compositions d'activités

Nous avons proposé un algorithme de composition des activités composites qui repose sur un ensemble de fonctions dépendantes domaine. Nous avons étudié la validité des activités résultantes de ces compositions. Nous nous intéressons à présent à d'autres propriétés qui peuvent être attendues de la composition des activités. Nous nous intéressons ici uniquement à des propriétés générales. Des propriétés spécifiques aux applications telles que celles présentées dans les travaux relatifs aux compositions d'aspects, ne sont pas étudiées ici[DFQJ08, PDS05].
Pour chaque propriété, nous l'énonçons puis identifions au niveau de l'algorithme les fonctions qui jouent sur le respect de ces propriétés. Le symbole † nous permet de mettre ces fonctions en évidence.
Nous illustrons ces propriétés sur le domaine fil rouge.

## 4.1 Idempotence

Nous déclinons la propriété d'idempotence sur l'opération de composition d'activités composites d'un point de vue *opération idempotente* et d'un point de vue *activité idempotente*.
Nous considérons tout d'abord l'opération de composition comme unaire sur un ensemble d'activités composites ; respecter la propriété d'idempotence consiste alors à vérifier la propriété suivante :

**Propriété sur Domaine 5 (Idempotence de l'opération de composition)** *La composition d'activités composites est idempotente si :*
$\forall Set \in \mathcal{CA}^{+}, \oplus_{\mathcal{F}} Set \equiv_{\sigma} \oplus_{\mathcal{F}} \{\oplus_{\mathcal{F}} Set\}$

Cette propriété est en particulier vérifiée si la fonction de détection des paires de fusion garantit qu'il n'existe pas de paires de fusion au sein d'une activité composite.

**Propriété sur Domaine 6 (Pas de paires de fusion dans une activité composite)** *Une fonction de détection des paires de fusion garantit l'idempotence de la composition d'activités composites si :* $\forall ca \in \mathcal{CA}$, *telle que* $isInDomain_{D}(ca)$

$$detectMergePairs(activities(ca), precedenceRelation(ca), conditionRelation(ca)) = \emptyset$$

Notons que par construction de l'algorithme de composition, une activité composite résultante d'une composition d'activités ne peut contenir des paires de fusion que si la normalisation sur domaine en introduit.
Une propriété plus intéressante est de vérifier que la composition de la même activité composite plusieurs fois renvoie l'activité composite elle-même.

**Propriété sur Domaine 7 (Idempotence d'une activité composite pour la composition)** *Une activité composite $ca$ est idempotente pour la composition d'activités composites si :*
$\oplus_{\mathcal{F}}\{ca, ca\} = ca$

**Exemple Fil rouge: Idempotence**

La propriété 1 du domaine fil rouge, page 52, assure que les éléments des ensembles de fusion appartiennent tous à des activités composites différentes, ce qui nous garantit l'idempotence de la composition d'activités composites dans ce domaine, la normalisation sur domaine n'introduisant pas de nouvelles activités pivots.
Par contre individuellement, seules les activités composites composées d'une seule activité *proceed* sont idempotentes.

## 4.2 Préservation des propriétés

La préservation des propriétés vise à assurer que lors d'une composition d'activités composites, des activités ou des éléments de relation ne disparaissent pas.

**Propriété 22 (Préservation des propriétés)** *Soient $n$ activités composites $ca_i$ telles que la composition est valide :*
$\oplus_{\mathcal{F}}{}_{i=1}^{i=n} ca_i = ca$ *avec* $ca_i = (id_i, AL_i, R_{p_i}, R_{c_i})$, $ca = (id, AL, R_p, R_c)$.
*Il y a préservation des propriétés si les trois points suivants sont vérifiés :*

**préservation des activités** : $\forall i \in 1..n, AL_i \subseteq_\sigma AL$ .

**préservation des relations de précédence** : $\forall i \in 1..n, R_{p_i|_\sigma AL} \subseteq_\sigma R_p$.

**préservation des relations de condition** : $\forall i \in 1..n, R_{c_i|_\sigma AL} \subseteq_\sigma R_c$.

La restriction dans l'expression de la préservation des relations de précédence et de conditions permet de vérifier ces propriétés alors que la préservation des activités n'est pas vérifiée. Ainsi seulement pour les activités "préservées" nous vérifions que les relations ont également été préservées.
La préservation des relations permet de vérifier la cohérence dans l'usage d'une activité dans des compositions différentes. Cette propriété est très utile pour assurer la cohérence des compositions. Par exemple, si dans une activité composite, l'activité d'initialisation a lieu avant une activité de calcul, il est probable que la disparition d'une précédence entre l'initialisation et l'activité de calcul engendre des erreurs.
Notons que le non-respect de la préservation des activités n'est généralement pas énoncé comme un problème dans les compositions par aspects, qui autorise, par exemple en omettant le *proceed*, la non exécution de l'activité principale, ou dans la composition de scénarii qui n'interdit pas de substituer à un événement d'autres évènements [KHJ06].

Dans [HKR+07], la préservation des propriétés lors de la composition de modèles est définie dans les termes de la sémantique des modèles. Si nous l'adaptons à notre définition

avec une sémantique *(notée sm)* basée sur les ensembles des traces d'exécution correspondant à une activité composite (cf. page 33), nous écrivons : $\forall(g_1..g_n) \in \mathcal{CA}, \bigcup_{i=1}^{i=n} sm(g_i) \subseteq sm(\bigoplus_{i=1}^{i=n}(g_i))$. L'inclusion d'une trace *t1* dans une trace *t2* s'entend ici comme la présence de la séquence d'actions correspondant à *t1* dans la trace *t2*, de nouvelles activités pouvant s'être interposées, par exemple des tests ; cette définition autorise donc la création de nouvelles activités. Nous n'en donnons pas ici une définition plus formelle car elle n'apporte pas de nouveaux éléments. Pour plus d'information sur les travaux sur les analyses des traces de workflow, voir par exemple [GBG05, vdAvDH+03].

La propriété de préservation des propriétés est très souvent recherchée, même au détriment de l'idempotence. Dès que la composition fait appel à des remplacements ou des retraits d'activités, la préservation des propriétés est perdue[KFJ07].

## Préservation et composition

**Copie :** par la propriété 14, l'équivalence entre les activités composites et leur copie assure la préservation des propriétés entre les activités composites et celles qui font l'objet de la composition.

**Détection des ensembles de fusion :** La détection des ensembles de fusion ne joue pas sur la préservation des propriétés.

**Fusion, Application des transformations et Restriction † :** L'impact de la fusion sur la préservation des propriétés passe par les transformations générées. Selon les transformations, il y aura perte de propriétés si des transformations retirent des activités de l'ensemble ou des relations sans que celles-ci aient été ajoutées sous la forme de nouvelles activités ou relations équivalentes. Ce point doit donc être vérifié si cette propriété est voulue.

De même avant d'opérer la normalisation, les relations résultantes sont restreintes aux activités restantes. En conséquence, la restriction n'occasionne pas de perte d'éléments de relations si chaque élément de relation retiré a bien son équivalent dans les éléments ajoutés.

**Normalisation partielle et sur domaine† :** Il convient de vérifier que les fonctions de normalisation respectent la préservation des propriétés.

Notons qu'en l'absence de paires de fusion, l'activité composite résultante fait référence à tous les éléments des activités composites initiales (activités et relations) ; elle est donc une extension des activités composites initiales [BBF+06].

`Exemple Fil rouge:` **Préservation des propriétés**

**préservation des activités** Lors de la fusion, la nouvelle activité créée est équivalente par construction à toutes les activités qu'elle fusionne. Les seules activités retirées sont les activités pivots, qui sont donc bien remplacées par une activité équivalente.

**préservation des relations de précédence et de conditions** Seuls les éléments de relation mettant en jeu des activités fusionnées sont modifiés par restriction. Or les éléments équivalents mettant en jeu l'activité créée ont été ajoutés. Il y a donc bien équivalence des relations.

Par définition le résultat de la normalisation est équivalent à l'activité normalisée. En conséquence, dans le domaine Fil Rouge la composition des activités composites préserve les propriétés.

## 4.3 Absence de duplication

Dans [BBF+06], les auteurs définissent l'opération de fusion comme préservant les propriétés sans duplication des informations au sens où un "même" élément du modèle n'apparaît pas plusieurs fois. La notion de même activité au sens d'une égalité portant sur l'identifiant de l'activité n'apporterait rien puisque par principe aucune activité n'est "dupliquée" : toute activité créée par copie a un nouvel identifiant. Vérifier qu'une composition ne créée pas d'activités dupliquées trouve son sens si on vérifie que la nouvelle activité composite ne contient pas des activités "dupliquant" un comportement.
Nous définissons donc l'absence de duplication dans le cadre des compositions d'activités comme suit.

**Propriété sur Domaine 8 (Absence de duplication)** *Une opération de composition d'activités ne duplique pas d'activités si étant données n activités composites* $ca_1..ca_n$ *telles qu'elles ne contiennent pas d'activités dupliquées, y compris entre elles* [1]*, alors* $\oplus_{i=1}^{i=n} ca_i$ *ne contient pas d'activités dupliquées.*
*S'il existe des activités équivalentes mais non dupliquées relativement à l'union des activités simples et des relations, nous considérons que l'opération de composition des activités ne crée pas de duplication si les activités dupliquées dans l'activité composite finale mettent uniquement en jeu des activités qui étaient équivalentes initialement.*

### Duplication et composition

**Copie :** par la propriété 14, l'équivalence entre les activités composites et leur copie assure que s'il n'y a pas d'activités dupliquées dans les activités composites initiales, il n'y en a pas dans la copie.

**Détection des ensembles de fusion :** les fonctions de détection des paires de fusion et de fusion des activités simples (cf. propriété 21) ne modifient pas les ensembles d'activités et de relations.

**fusion, application des transformations et restriction † :** il y aura possibilité de duplication d'une activité, dans les cas suivants :

1. deux activités sont considérées comme dupliquées dans un ensemble d'activités composites, si elles sont équivalentes et que les relations dans lesquelles elles interviennent sont équivalentes (cf. Propriété 11)

a) la fusion ou une transformation crée une nouvelle activité simple "duplication" d'une activité existante sans retirer cette dernière (pas nécessairement une activité pivot) ;

b) la fusion crée de nouvelles activités, duplications les unes des autres.

La restriction intervient, elle, dans la détermination de l'absence de duplication si les relations retirées avaient été utilisées pour différencier les activités.
La duplication des activités peut cependant être éliminée par une nouvelle fusion des activités dupliquées si celles-ci sont identifiées comme ensemble de fusion, ou par modification des relations. Il convient donc pour prouver cette propriété de éventuellement s'assurer de cette dernière possibilité.

**Normalisations sur domaine † :** Les fonctions de normalisation peuvent créer des activités dupliquées par ajout d'activités ou modification des relations de conditions s'il existait des activités équivalentes. Lorsque la non-duplication est recherchée, il convient de vérifier cette propriété au niveau de la normalisation.

`Exemple Fil rouge:` **Absence de duplication**

**retrait des activités fusionnées** Lors de la fusion, les activités pivots dont est issue toute nouvelle activité sont retirées de l'ensemble des activités (transformations *remove*).

**création uniquement d'activités pivots** Les seules activités créées sont des activités pivots qui sont assujetties à l'ensemble des éléments de précédence et de condition des activités fusionnées. Or ces activités sont par construction exclusives, sinon elles auraient appartenues au même espace de fusion et la fusion dans le domaine fil rouge préserve les conditions.

**restriction** L'absence de duplication a été démontrée jusqu'ici sans tenir compte de relations qui pourraient être ôtées par restriction. En conséquence la restriction n'intervient pas sur la duplication.

**normalisation** Si nous considérons que les activités composites initiales ne contiennent pas d'activités équivalentes en dehors de pivots, nous avons montré que l'activité composite résultante avant normalisation ne contient pas d'activités dupliquées. La propriété 16 assure seulement que lorsqu'une activité composite ne contient pas d'activités dupliquées, il n'y a pas de duplication parmi les activités normalisées dans l'activité résultante. Par construction, les seules activités normalisées sont les successeurs des activités pivots. En conséquence, elles ne sont pas dupliquées entre elles, ni avec d'autres puisque par hypothèse il n'existe pas d'autres activités qui leur soient équivalentes.

## 4.4 Indépendance de l'ordre des activités

**Propriété sur Domaine 9 (Composition indépendante de l'ordre)** *Il y a indépendance dans la composition d'activités composites si :*
$\forall ca_i \in \mathcal{CA}^+$, $\bigoplus_{D\,i=1}^{i=n} ca_i$ *donne un résultat équivalent quelque soit l'ordre des activités, en particulier si un ordre de composition échoue, toutes les compositions échouent.*

### Indépendance des ordres de composition et composition

**Copie :** Elle n'entre pas en jeu dans la vérification de cette propriété puisque la copie vérifie l'équivalence entre les activités composites et leur copie (cf. propriété 14).

**Détection des ensembles de fusion † :** Le calcul des ensembles de fusion ne dépend pas de l'ordre des paires ; il convient néanmoins de s'assurer que *le calcul des paires de fusion ne dépend pas de l'ordre des activités.*

**Fusion † :** La propriété 2 doit être vérifiée pour que la composition des activités composites ne dépendent pas de l'ordre.

**Application des transformations et restriction † :** Les fonctions de détection des ensembles de fusion et la fusion des ensembles résultants ne modifient pas les ensembles d'activités et de relations initiaux. Il n'y a donc pas d'interactions entre les fusions simples d'ensembles de fusion, et l'ordre dans les traitements n'intervient donc pas[2]. Les ajouts d'activités et d'éléments de relation se font par union, il n'y a donc pas de dépendance avec l'ordre des activités. Il y a donc indépendance de l'ordre des activités lors de l'application des transformations si l'application des transformations sur les activités résultantes ne depend pas de l'ordre des activités (cf. Propriété 1).

**Normalisation Partielle† :** Il convient de démontrer qu'elle ne dépend pas de l'ordre des activités résultantes.

**Itération :** Le procédé étant réitéré soit il n'y a plus de couple de fusion et il y a donc équivalence des résultats, soit nous recommençons par le même procédé dont nous avons montré qu'il conduit sous certaines conditions à des résultats équivalents.

**Normalisation sur domaine† :** Elle ne doit pas dépendre de l'ordre des activités.

**Remarques :** Cette analyse sous-tend que la fusion est déterministe, c-à-d. qu'avec un même ensemble de fusion ce sont toujours les mêmes transformations et activités qui sont créés, en particulier en faisant abstraction des autres actions de fusions qui ont été opérées. Dans un procédé interactif, ce point ne sera plus vrai, le programmeur pouvant anticiper sur les modifications opérées ou sélectionner les ensembles de fusion de manière itérative.

**`Exemple Fil rouge`: Indépendance des ordres de composition**

- **Détection des ensembles de fusion indépendant de l'ordre :** vérifiée d'après la propriété 4 du domaine Fil Rouge ;
- **Fusion indépendante de l'ordre :** vérifiée d'après la propriété 8 du domaine Fil Rouge ;
- **Application des transformations indépendante de l'ordre :** vérifiée d'après la propriété 5 du domaine Fil Rouge.
- **Normalisation indépendante de l'ordre :** vérifiée d'après la propriété 15 du domaine Fil Rouge.

2. L'utilisateur pouvant être impliqué dans la résolution, il peut être préférable de lui présenter certains couples de fusion plus faciles à résoudre que d'autres mais au niveau du modèle, nous ne les prendrons pas en compte.

## 4.5 Associativité

La composition des activités peut intervenir dans des temps différents. A partir d'une activité composite obtenue par composition d'un ensemble d'activités, de nouvelles activités sont à nouveau composées. Peut-on alors garantir que le résultat est le même, à l'équivalence près, que si nous avions composé l'ensemble des activités ?

Cette propriété est particulièrement intéressante dans le cadre d'une évolution incrémentale de l'application. Les demandes de composition interviennent alors au fil du temps soit par un travail collaboratif où les différentes compositions visent à ajouter des capacités à un système selon des points de vues différents, soit par une évolution des activités composites, une activité composite étant modifiée au fil du temps par l'ajout de nouvelles activités.

Nous recherchons donc la propriété d'associativité qui s'énonce ainsi :

**Propriété 23 (Associativité de la composition)** $\forall Set_1 \in \mathcal{CA}^+, \forall Set_2 \in \mathcal{CA}^+$
$\bigoplus_D\{Set_1 \cup \{\bigoplus_D\{Set_2\}\}\} = \bigoplus_D\{Set_1 \cup Set_2\} = \bigoplus_D\{\{\bigoplus_D\{Set_1\}\} \cup Set_2\}$

Lorsque $\bigoplus_D$ est indépendante de l'ordre des activités, il suffit de démontrer que la propriété d'associativité est vraie pour 3 éléments :
$\bigoplus_D\{ca_1, \bigoplus_D\{ca_2, ca_3\}\} = \bigoplus_D\{ca_1, ca_2, ca_3\} = \bigoplus_D\{\bigoplus_D\{ca_1, ca_2\}, ca_3\} = \bigoplus_{D}{}_{i=1}^{i=3} ca_i$

En effet, sous cette hypothèse, supposons que la propriété d'associativité soit vraie pour n-1, elle sera vraie pour n activités, n>3, car :
$\bigoplus_D\{ca_n, \bigoplus_{D}{}_{i=1}^{i=n-1} ca_i\} =$ *(d'après l'associativité à n-1)*
$\bigoplus_D\{ca_n, \bigoplus_D\{ca1, \bigoplus_{D}{}_{i=2}^{i=n-1} ca_i\}\} =$ *(d'après l'associativité à 3)*
$\bigoplus_D\{\bigoplus_D\{ca_n, ca1\}, \bigoplus_{D}{}_{i=2}^{i=n-1} ca_i\}$

La vérification de cette propriété met en jeu de nombreux facteurs que nous n'avons pas su identifier dans un cadre général. Nous la démontrerons donc dans chaque domaine. Néanmoins, nous pouvons identifier plusieurs propriétés qui interviennent dans le respect de l'associativité :
- la préservation des activités, en particulier les activités "pivots", favorise l'associativité puisque la composition opère sur les mêmes éléments, en particulier la détection des pivots ;
- l'associativité de la fonction $\sum_D$ sur les transformations même si elle n'est pas suffisante, semble nécessaire à l'associativité de la composition des activités.

**`Exemple Fil rouge:` Associativité**

Nous avons démontré l'indépendance des ordres de composition. Nous démontrons l'associativité pour 3 activités composites.
**Erreurs de composition :** Nous commençons par éliminer les cas d'erreurs : le respect de l'associativité nécessite que les erreurs se produisent quelque soit la manière de composer. Des erreurs se produisent lors de la composition si :
- les activités pivots n'ont pas la même arité *(échec de* $detectMergePairs$*)* ;
- l'activité résultante n'est pas déterministe *(échec de* $normalizeOnDomain$*)*
La fusion des activités pivots ne modifiant pas l'arité des pivots, si la première erreur est levée, lors de la composition de 2 activités composites, l'erreur se produira quelques soient les compositions précédentes.
La seconde erreur suppose que dans le résultat d'une composition nous avons trouvé deux activités qui accèdent à une même variable dans un ordre non déterminé au moins une fois en écriture. La composition dans le domaine fil rouge garantissant la préservation des relations et des activités, nous sommes certains que quelque soit l'ordre des compositions cette erreur se reproduit.
**Composition sans erreur :**
Soient $ca_1, ca_2, ca_3$, 3 activités composites telles que $ca_i = (id_i, AL_i, R_{p_i}, R_{c_i})$ où
$AL_i = \{a_1^i, ...a_{n_i}^i\}$ avec $pr_1^i..pr_{k_i}^i$ les activités pivots dans $AL_i$.
$Activities = \bigcup_{i=1}^{i=3} AL_i$, $R_p = \bigcup_{i=1}^{i=3} R_{p_i}$, $R_c = \bigcup_{i=1}^{i=3} R_{c_i}$ ;
Les ensembles de fusion pour $(Activities, R_p, R_c)$ sont :
$\{\{pr_1^1, pr_1^2, pr_1^3\}, \{pr_1^1, pr_2^2, pr_1^3\}, ...\} = \{s^{111}, s^{121}, s^{131}, ....s^{k_1k_2k_3}\}$.
$\forall s^k, mergeSimpleActivities_{FilRouge}(Activities, R_p, R_c, s^k)$
$= (pr^k, newR_p^k, newR_c^k, todoR^k)$ où
$pr^k$ est une nouvelle activité *proceed* obtenue par fusion des *proceed* dans $s^k$,
$newR_p^k$ définit les relations de précédence avec la nouvelle activité,
$newR_c^k$ définit l'union des gardes portées par toutes les activités pivots de $s_k$ et les associe à la nouvelle activité.
$todoR^k$ est l'ensemble des transformations composé des substitutions qui formeront l'unificateur, tandis que les autres transformations consistent à retirer les activités composant l'ensemble de fusion.
L'application de toutes les transformations obtenues par fusion donne une nouvelle activité composite dans laquelle toutes les activités pivots ont été remplacées par de nouvelles, résultats de l'unification.

L'associativité sera vérifiée si en appliquant la fusion sur $ca_1$ et $ca_2$ puis $ca_3$ on obtient le même résultat. Les espaces de fusion obtenus en composant $ca_1$ et $ca_2$ sont : $\{\{pr_1^1, pr_1^2\}, \{pr_1^1, pr_2^2\}, ...\{pr_{k_1}^1, pr_{k_2}^2\}\} = \{s^{11}, s^{12}, ....s^{k_1k_2}\}$.
Le résultat de la fusion est alors composé d'activités pivots résultante de l'unification des pivots : $pr_{ij}^{12}$ exprime l'activité pivot résultante de la fusion entre $pr_i^1, pr_j^2$. Chacune de ces activités a pour prédécesseurs et conditions, l'union des prédécesseurs et conditions des activités $pr_i^1$ et $pr_j^2$.
L'application d'une nouvelle composition avec $ca_3$ crée les ensembles de fusion constitués du résultat de la fusion des pivots entre ($ca_1$ et $ca_2$) et les pivots de $ca_3$ : $\{(pr_{11}^{12}, pr_1^3), ...(pr_{k_1k_2}^{12}, pr_{k_3}^3)\}$
Le résultat de la fusion entre $pr_{ij}^{12}$ et $pr_l^3$ crée une activité $pr_{ijl}^{123}$ équivalente à la fusion entre $pr_i^1, pr_j^2, pr_l^3$ car l'unification des variables est associative, et les unions de précédence et de conditions sont également associatives. La normalisation sur domaine n'ayant aucune action, elle n'intervient dans l'associativité que en cas d'erreur. La normalisation partielle ne duplique pas d'activités pivots et n'intervient donc pas non plus dans l'associativité.

# CHAPITRE 5 Conclusion : Eléments pour la composition d'activités

Ce chapitre sert de fil à conducteur à l'écriture d'applications définissant de nouveaux domaines de composition d'activités. Il est un résumé de cette partie.
Il discute pour chaque point de l'algorithme les limites et les contraintes, ainsi que les perspectives.

## 5.1 Modélisation et domaine d'application

Nous avons défini un métamodèle dit MM4CA(cf. figure 2.1) qu'il est nécessaire de raffiner en fonction des domaines d'application. Les points plus spécifiquement à raffiner sont :

**Information** Les informations caractérisent les données que peut porter une activité. Une fonction d'équivalence sur *information* doit être fournie.

**Variable** Les variables peuvent être spécialisées pour porter des informations telles que le type, la visibilité, ... Une fonction d'équivalence sur *variable* doit être fournie.

**Contraintes :** A cette structure, il peut être nécessaire d'ajouter des contraintes précisant les conditions sous lesquelles les activités composites dans le domaine d'application sont bien formées.
Ces contraintes s'accompagnent de la définition de deux fonctions sur domaine :
$isInDomain_D$ (cf. fonction 1 sur domaine, page 40) et $normalizeOnDomain_D$ (cf. fonction 2 sur domaine, page 46).
Cette dernière permet de rendre une activité composite valide conforme aux contraintes relatives au domaine d'application.

### 5.1.1 Mise en conformité : limites et perspectives

Notre approche est fondée sur le métamodèle d'activités MM4CA(cf. figure 2.1). Ce métamodèle définit en particulier les structures de contrôles comme des relations entre les activités. Nous avons donc émis comme hypothèse que dans le cadre de composition d'activités composites, nous étions toujours capable de nous ramener à cette modélisation.
Les domaines d'application ont généralement leur propre métamodèle qui ne correspond pas directement à cette modélisation. Il sera donc nécessaire d'exprimer des transformations entre les métamodèles applicatifs et le métamodèle MM4CA, et réciproquement.

L'*extension* du métamodèle MM4CA à la prise en compte d'autres éléments de contrôle, comme la gestion des exceptions (try-catch, compensation, ...) et leur prise en compte dans la composition est une de nos perspectives.
L'intension portée par certaines formes de contrôles peut également être perdue dans la transformation vers un modèle conforme à notre métamodèle si des mécanismes de trace ne sont pas mis en place. Par exemple en BPEL on peut utiliser aussi bien une séquence qu'un lien pour exprimer la précédence entre deux activités. Le lien peut être nommé et ainsi véhiculer de la "sémantique"(cf. 7.4.2.4). Cette information est perdue dans notre approche aujourd'hui. La représentation choisie des activités dupliquent les activités dont l'exécution suppose la fin d'activités

## 5.2 Composition et Interprétation

La composition des activités repose sur la définition d'interprétations, c-à-d. la donnée de fonctions sur domaine. Selon les propriétés vérifiées par ces fonctions, l'algorithme de composition vérifie ou non les propriétés que nous avons définies au §4.
Nous rappelons ici les fonctions sur domaine qu'il s'agit de définir.

### 5.2.1 Détection des paires de fusion et construction des ensembles de fusion

La fonction 3 sur domaine : $detectMergePairs_D$, page 49, détermine les activités qui forment des paires de fusion.
- Elle peut lever l'erreur *conflit de pivots*.
- Une des propriétés souvent attendue de cette fonction est de ne pas dépendre de l'ordre des activités (cf. propriété 20 sur domaine).
- Le calcul des paires de fusion puis des ensembles de fusion peut correspondre à des temps de calcul important en fonction des activités pivots, de la complexité de leur détection, de leur nombre, ... Il paraît important d'établir lors de la définition d'un domaine ces différentes cardinalités et la complexité des algorithmes.

Dans nos expériences, le nombre d'activités pivots est très inférieur aux nombres d'activités. De plus les activités pivots peuvent être organisées en sous paquets dans lesquels des paires peuvent intervenir (par exemple, la nature des activités est utilisée dans la composition des orchestrations pour déterminer les paires de fusion), réduisant ainsi la complexité.

### 5.2.2 Transformations et composition des transformations

La composition des activités utilise des transformations des activités simples et des relations. En fonction des domaines, ces transformations peuvent être plus ou moins riches. La fonction 4 sur domaine : $\sum_D$, page 54, détermine la manière dont les transformations sont composées.
- Cette fonction peut lever l'erreur *conflit de transformation*.
- Une des propriétés qui peut être attendue de cette fonction est de ne pas dépendre de l'ordre des transformations (cf. propriété 1 sur domaine, page 54). Cette propriété est nécessaire à l'indépendance de l'ordre dans la composition des activités composites (cf. propriété 9, page 67).

### 5.2.3 Fusion d'activités simples

La fonction 5 sur domaine : $mergeSimpleActivities_D$, page 56, détermine à partir d'un *ensemble de fusion* (i) les activités à créer, (ii) les relations de précédence et de conditions associées à ces activités et (iii) les transformations à appliquer.
Cette fonction peut lever l'erreur *conflit de fusion*.
Une des propriétés qui peut être attendue de cette fonction est de ne pas dépendre de l'ordre des activités (cf. propriété 2 sur domaine, page 57). Cette propriété est nécessaire à l'indépendance de l'ordre dans la composition des activités composites (cf. propriété 9, page 67).
La non-création de circuit ou d'incohérence sont deux autres propriétés (cf. propriétés sur domaine 3 et 4) exigées de cette fonction par la composition des activités composites.

### 5.2.4 Composition d'activités composites

Cette fonction dépend du domaine par les fonctions définies précédemment. Il est important de noter que le nombre des ensembles de fusion doit décroître entre deux tours de boucle et que l'application des transformations doit assurer la validité du graphe d'activités obtenu (cf. propriétés exigées, page 60).

## 5.3 Propriétés

Nous avons identifié plusieurs propriétés issues de nos expériences ou de la littérature dont nous avons précisé la sémantique relativement à notre formalisation.
Un point important de ce travail est l'identification des activités comme étant "les mêmes". Nous nous basons pour la vérifier sur l'équivalence des activités et l'équivalence des relations. Les différentes propriétés qui suivent sont fortement liées à ces notions.

### 5.3.1 Idempotence

Lors de la composition d'activités composites, que se passe-t-il si l'on cherche à recomposer le résultat d'une composition ? Nous pouvons ainsi décliner cette propriété au niveau de l'opération de composition et dans ce cas la propriété est vraie pour toute activité ou au niveau de quelques activités qui vérifient cette propriété.

**Remarques** En fonction du domaine d'application ces propriétés seront ou non recherchées. En effet, la composition d'une même activité plusieurs fois peut être voulue pour multiplier des comportements alors que dans d'autres cas, en particulier en présence de développement collaboratif et incrémentiel, on préférera que le résultat de la composition ne mette pas en jeu la même activité plusieurs fois.
Ainsi la vérification de cette propriété n'est pas indépendante de la préservation des propriétés et de l'absence de duplication, mais a des objectifs différents.

**Éléments à vérifier** L'idempotence de la composition d'activités composites repose sur la définition des activités composites et la fonction de détection des paires de fusion qui doit alors respecter la propriété sur domaine 6.

### 5.3.2 Préservation des propriétés

Lors de la composition d'activités composites, on peut vouloir garantir que des activités et/ou que les relations de précédences et de conditions ne disparaissent pas.

**Éléments à vérifier** La préservation des propriétés dépend de la fusion et de l'application des transformations. Il s'agit de s'assurer que celles-ci ne retirent pas d'activités sans en ajouter d'équivalentes, et ne suppriment pas de relations sans en ajouter d'équivalentes.

### 5.3.3 Absence de duplication

Lors de la composition d'un ensemble d'activités composites, il apparaît nécessaire de garantir que l'activité résultante ne contient pas d'activités dupliquées sauf bien sûr si des activités étaient déjà dupliquées dans l'ensemble initial. Or, il est possible que l'ensemble de départ contienne également des activités équivalentes mais non dupliquées. Dans ce cas, les activités équivalentes peuvent réapparaître comme dupliquées après composition.

**Remarques** Cette propriété est difficile à vérifier parce qu'elle suppose qu'en cas d'activités équivalentes présentes initialement ou introduites, la composition est la capacité de les fusionner à nouveau ou d'interdire leur introduction. Notons que dans ce cas, rechercher l'absence de duplication revient à rechercher l'idempotence quelque soient les activités composites.

**Éléments à vérifier** L'absence de duplication doit être vérifiée en étudiant la fusion et l'application des transformations.

### 5.3.4 Indépendance de l'ordre des activités

Lorsque l'on compose un ensemble d'activités composites, en fonction du domaine, il peut être intéressant de garantir que le résultat de la composition ne dépend pas de l'ordre des activités.

**Éléments à vérifier** Cette propriété est vérifiée si la détection des paires de fusion et la fusion d'ensemble de fusion ne dépendent pas de l'ordre des activités (cf. fonction 2 sur domaine).
L'application des transformations sur les activités résultantes ne doit pas dépendre de l'ordre des activités (cf. fonction 1 sur domaine).

## 5.4 Suite du document

La modélisation proposée fournit donc un cadre à différentes formes de composition d'activités, ce que nous démontrons dans la partie suivante *application*.

*Deuxième partie*

# APPLICATIONS

## Dans cette partie...

Le travail de modélisation et de formalisation présenté dans la partie précédente a mis en exergue un processus de composition basé sur des fonctions dépendant du domaine applicatif. Nous l'étayons à présent au travers de son application à deux domaines applicatifs. Pour chacun de ces domaines, nous commençons par identifier le domaine et nos objectifs, puis le formalisons relativement aux éléments de formalisation précédemment établis. Nous explicitons alors les mises en œuvre de ces modèles avant de conclure chaque application par un retour sur expérimentations, une analyse de ces travaux relativement à l'état de l'art et nos perspectives.
Le premier chapitre (cf. §6) décrit la composition des interactions entre composants hétérogènes. Les interactions capturent les modifications d'échanges de message des composants à l'exécution, exprimées selon des points de vue différents. La composition des interactions vise à calculer un comportement cohérent des interactions entre composants. En particulier l'associativité des compositions sera recherchée.
Le second chapitre (cf. §7) s'intéresse à la composition des orchestrations au sens des architectures SOA. Dans ce contexte, les compositions interviennent sur plusieurs activités pivots simultanément et il n'est pas toujours possible de décider des fusions à opérer. L'objectif est d'aider le concepteur dans cette tâche difficile. Ce domaine nous permet d'aborder l'interactivité dans la composition des activités. Les propriétés à préserver contraindrons les possibilités de modification tout en guidant l'utilisateur dans le processus de composition. Nous établissons ainsi l'intérêt d'une démarche unifiée pour décrire ces différents algorithmes.

# CHAPITRE 6

# Compositions d'Interactions entre composants hétérogènes

*The architecture of a software system defines that system in terms of components and of interactions among those components [SDK+95].*

## Contexte de recherche et contributeurs

*Ce travail a été initié avec Anne-Marie Pinna-Dery. Il a fait l'objet de la thèse de Stéphane Ducasse [Duc97] sous ma co-direction puis Laurent Berger [Ber01] sous la direction d'Anne-Marie. L'intégration des dimensions composant et interopérabilité a été initiée par Michel Riveill. David Emsellem est le principal développeur des versions java et .Net du service d'interactions. Je renvoie le lecteur à [BFCE+04] et [BFEPDR04] pour une présentation rapide des interactions en terme de service et langage. Le langage d'expression des interactions (ISL : Interaction Specification Language) a évolué au fur et à mesure des applications. Ainsi Pascal Rapicault a contribué, sous ma direction, à son évolution au travers de son application à la gestion de la cohérence dans les canevas logiciels [Rap02]. Au delà des travaux présentés dans ce mémoire, qui sont pour l'essentiel nés des recherches autour de ce langage, une nouvelle perspective a été ouverte en appliquant le modèle sous-jacent au langage à la composition d'assemblages de composants [CFWBFT+05]. Enfin je noterais que la thèse de Audrey Occello [Occ06] dirigée par Anne-Marie Pinna-Dery, a ouvert un autre point de vue en s'intéressant à la cohérence des assemblages.*
*Dans ce mémoire, je propose une vision un peu différente en revisitant ces travaux relativement au métamodele* MM4CA *défini dans la partie I.*

## Plan de ce chapitre

Dans un premier temps (cf. §6.1), nous précisons nos objectifs et notre vocabulaire relativement à la composition dans le cadre d'interactions entre composants hétérogènes. Nous formalisons alors la composition d'activités dans le domaine des interactions et développons les propriétés respectées par cette composition (cf. §6.2). Nous utilisons cette formalisation pour expliciter la composition des "règles d'interactions" (cf. §6.3) avant de présenter plus précisément la mise en oeuvre de cette modélisation (cf. §6.4).

Enfin nous concluons ce chapitre par un retour sur expérimentation et le positionnement de notre travail vis-à-vis d'autres travaux, ce qui nous amène à expliciter nos perspectives relativement à ce travail (cf. §6.5).

## 6.1 Domaine applicatif et objectifs

L'omniprésence de l'informatique et des réseaux, en même temps que la démocratisation de l'usage des dispositifs logiciels rend indispensable l'adaptation des applications en cours d'exécution. Entre autres causes d'adaptations non-prévisibles, nous trouvons la découverte ou disparition de composants, la modification de l'environnement d'exécution, la prise en compte du " profil " utilisateur,etc.[BF06]. La mise en œuvre de ces adaptations, même sans aborder le côté autonomique [PB05] suppose de résoudre certains défis.

**Hétérogénéité des composants et expression des adaptations** Ces adaptations mettent en jeu des composants hétérogènes [1] et distribués et peuvent nécessiter un contrôle des infrastructures [MDBF06]. Il s'agit donc d'être capable d'expliciter les adaptations indépendamment des plates-formes, tout en autorisant une mise en œuvre efficace [Nan04, BFCE+04].

**Séparation des préoccupations et compositions** L'intensification des usages de dispositifs logiciels et la complexité des applications nécessitent des adaptations exprimées par différents acteurs, experts à l'égard de préoccupations différentes. De fait, il est donc nécessaire d'offrir un support à la définition séparée des adaptations et des mécanismes de composition qui ne forcent pas une connaissance mutuelle des adaptations. Ce constat induit à la fois des compositions indépendantes de l'ordre des adaptations et la détection des conflits éventuels [FF05, PSDB04, DFS04, DFL+05]. L'invalidité d'une composition peut impliquer le rejet d'une ou plusieurs adaptations. Il convient donc d'être capable à la fois d'adapter le système en introduisant de nouveaux composants mais également en retirant des adaptations. Formellement, le retrait d'adaptation est une forme d'adaptation indispensable au contrôle de l'évolution d'une application.

**Sûreté des adaptations** Dans un contexte dynamique d'adaptation, il est particulièrement pénalisant d'introduire des incohérences qui peuvent conduire à un comportement incohérent de l'application voir à son arrêt. Il apparaît donc essentiel de corréler aux mécanismes d'adaptations des moyens pour garantir la sûreté de fonctionnement des applications [BLMD06, FTS06, Occ06].

En conclusion, les adaptations dynamiques doivent faire face à une grande complexité technique pour répondre aux objectifs de déclarativité et d'interopérabilité. Elles doivent pouvoir être exprimées séparément, permettant ainsi à chaque acteur d'adapter un ensemble de composant, même en l'absence d'une connaissance globale des assemblages. Cependant, le comportement de l'application adaptée doit rester cohérent et déterministe.

1. i.e. reposant sur différents modèles et plates-formes d'exécution : Fractal, EJB, Services Web, CCM.

Nous proposons donc de mêler les approches par modèles qui donnent la liberté d'expression et l'indépendance plate-forme, les techniques de mises en œuvre qui rendent le système opérationnel, des outils de compositions qui reposent sur des théories bien fondées pour assurer la validité des adaptations à l'exécution.
Avant d'expliciter davantage notre approche, nous posons à présent le vocabulaire propre à ce domaine d'application.

### 6.1.1 Des adaptations de composants aux interactions logicielles

Nous considérons plus spécifiquement les adaptations portées par les interactions entre les composants.

**Composants** Donner une définition du concept de composant logiciel n'est pas notre objectif. La définition précise qui pourrait être donnée des composants CCM [obj05] ou des composants EJBs [DÜYK01] est très différente, les premiers intégrant à la fois une dimension de pré-requis et d'assemblage que les deuxièmes n'abordent pas. Nous nous contenterons donc de la définition générale de Szyperski [SM02] : "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component is deployed independently and is subject to composition by third parties*". En terme de structure, UML définit un composant comme dérivant de "classifier" dont dérive également le concept de classe, à l'inverse dans ArchJava [ACN02] les composants eux-mêmes sont les instances de classes de composants. Afin de distinguer quand cela nous semble nécessaire le composant "modèle" et les instances qui lui "correspondent" à l'exécution, nous parlerons également de classes de composants et de composants pour les instances. Peu nous importe la mise en œuvre en de multiples objets de cette instance, c'est l'entité référençable qui constitue pour nous le composant. L'exécution d'un système à composants consiste à réagir à des "*messages*" (éventuellement événements) émis par des composants vers d'autres composants. L'*interface* d'un composant est décrite par les *opérations* qu'il fournit et qu'il requiert. Le *comportement* d'un composant se caractérise à la réception d'un message par les valeurs de retour et les messages émis vers d'autres composants.

**Interactions** "L'interaction dans le langage courant signifie l'action ou l'influence réciproque qui peut s'établir entre deux objets ou plus... Une interaction est toujours suivie d'un ou plusieurs effets. En physique, en chimie ou en biologie, une interaction a pour effet de produire une modification de l'état des objets en interaction, comme les particules, atomes ou molécules." De la même manière, une *interaction* entre des composants modifie leur comportement. Prenons l'exemple d'une application à base d'agendas collaboratifs (cet exemple sera étayé en section 6.1.2) pour expliciter comment les interactions impactent le comportement des composants, par :
- la mise en place de contrôle : l'exécution de certaines opérations d'un composant est dépendante de conditions sur les composants en interaction. *L'ajout d'un rendez-vous sur un agenda de groupe peut être conditionné par un accord d'une majorité des membres du groupe.*
- l'émission de nouveaux messages : l'exécution de certaines opérations d'un composant implique l'exécution d'opérations sur les composants en interaction. *Une interaction de notification entre l'agenda de Doc et un afficheur consiste à modifier le comportement de l'agenda pour que l'afficheur reçoive un ordre d'affichage chaque*

*fois qu'un rendez-vous est ajouté ou retiré dans l'agenda.*

Notons dès à présent qu'à la différence des travaux sur les connecteurs [Car03, BP01], nous nous intéressons ici à expliciter des interactions qui ne font pas partie du composant et n'expriment donc pas des connexions entre services fournis et requis, mais entre services fournis uniquement.

Une interaction modifie le comportement du composant et non pas les classes de composants. *Ainsi, l'ajout d'interactions de notification entre l'agenda de Mireille et l'afficheur ne concerne que l'agenda de Mireille et non pas tous les agendas.*

**Schéma d'interactions** Un schéma d'interactions décrit les modifications de comportement qu'il faudra apporter aux composants qui seront liés par ce schéma. Les différentes modifications de comportement des composants ainsi décrites sont dans la suite appelées des *règles d'interactions*. *Par exemple, un schéma d'interaction "observateur-observable" exprime que le comportement d'un composant (observable) devra être modifié en terminant l'exécution de toutes ses opérations par une notification à un composant (observateur).*

L'application d'un schéma d'interactions à des composants donnés conduit à la création et à la mise en place d'une *instance du schéma d'interactions* ou plus simplement d'interactions dans la suite. Une interaction ne modifie pas pareillement le comportement de tous les composants liés. Les modifications dépendent du rôle du composant dans l'interaction. *Ainsi une interaction entre un agenda d'équipe et un afficheur partagé contrôle les accès sur l'agenda personnel pour en notifier l'afficheur, tandis qu'une autre interaction contrôle les accès à l'agenda de l'équipe pour répercuter les ajouts et retraits de rendez-vous dans l'agenda personnel.*

La *cohérence* d'une interaction est dite vérifiée lorsque les modifications de comportements des composants liés sont effectives. *Ainsi, si l'exécution d'une opération d'un composant observable en interaction avec un composant observateur ne se termine pas par une notification de ce dernier, la cohérence de l'interaction instance du schéma observateur-observable n'est pas vérifiée.*

Un *graphe d'interactions* est un hypergraphe dont les nœuds sont les composants et les arêtes les interactions.

**Pose et retrait d'interactions** L'instanciation d'un schéma d'interactions avec pour paramètre les composants à lier consiste à créer des interactions et à modifier le comportement de ces composants pour que la cohérence des interactions soit vérifiée. Nous parlons alors de la pose d'interactions par instanciation de schémas d'interactions.
Le retrait d'interactions consiste à retirer les modifications de comportements que la pose des interactions avait occasionnées.
En cours d'exécution, un utilisateur est amené à créer des schémas d'interactions, un autre à poser et retirer des interactions. En fonction des applications, l'utilisateur adapte l'application à un contexte matériel ou logiciel changeant par la définition de schémas

et la pose et le retrait d'interactions. Dans certaines applications, des schémas d'interactions sont fournis avec l'application ; l'utilisateur final les utilise pour adapter son application [DBFA+01].

**Composition d'interactions** Un composant peut être lié par plusieurs interactions simultanément. La composition des interactions consiste à :
- déterminer les modifications de comportements des composants liés,
- vérifier que la cohérence de toutes les interactions dans lesquelles intervient le composant est vérifiée et à refuser la composition dans le cas contraire. Nous dirons que la cohérence locale à un composant est vérifiée si la cohérence de toutes les interactions liant ce composant est vérifiée.

En particulier, nous attendons de la composition des interactions qu'elle respecte les propriétés suivantes :
-**(p1)** elle n'ajoute pas d'attente et préserve les ordres explicités entre les actions ainsi que les conditions à leur exécution. En effet, les interactions explicitent des ordres et des conditions aux actions que nous tenons à préserver, sans ajouter d'ordre dû à la composition [2].
-**(p2)** elle ne dépend pas de l'ordre de pose des interactions et est associative si la composition est acceptée. En effet, supposons qu'un utilisateur introduise un contrôle qui vise à le notifier de l'exécution d'une opération. Puis un autre utilisateur demande que cette même opération ne puisse être exécutée que si une condition donnée est vérifiée. On peut s'attendre à ce que le premier utilisateur ne soit notifié que quand l'opération est effectivement exécutée et donc quand la condition est vraie. Le même comportement est attendu quelque soit l'ordre de pose des interactions ;
-**(p3)** la réversibilité : le retrait d'interactions doit être possible et donner le même résultat que si l'interaction n'avait jamais été posée.

### 6.1.2 Exemples de compositions d'interactions

Afin de donner une vue intuitive du domaine et de son utilisation, nous présentons un exemple simple et usuel de gestion d'agendas. Cette application est composée d'une classe de composants *Display* qui permet l'affichage de message, d'une classe de composants *SecurityManager* qui permet de valider des appels, d'une classe de composants simplifiée *Database* qui permet de mémoriser les agendas dans une table de hachage en fonction du nom de l'utilisateur de l'agenda et une classe de composants *Agenda*. En cours d'exécution, les instances de ces différentes classes de composants sont liées et déliées par des interactions pour s'adapter au contexte d'exécution (création d'un nouveau composant de groupe, ajout d'un membre dans l'équipe, nécessité de sécurité lors de la mise en réseau, ajout d'une base de données,. . . ). Ainsi certains composants *Agenda* sont rendus persistants, un agenda de groupe est créé par la pose d'interactions entre plusieurs agendas, un *authentificateur* est associé à certains agendas pour vérifier que seuls les utilisateurs autorisés accèdent à l'agenda. La figure 6.1 visualise un réseau de composants et d'interactions à un moment donné de l'exécution de l'application.

2. L'introduction d'un nouvel ordre entre des activités serait soit basé sur l'ordre des déclarations, soit basé sur l'utilisateur ce qui suppose une connaissance de toutes les interactions ; Ces différentes approches nous semblent contradictoires avec la séparation des préoccupations.

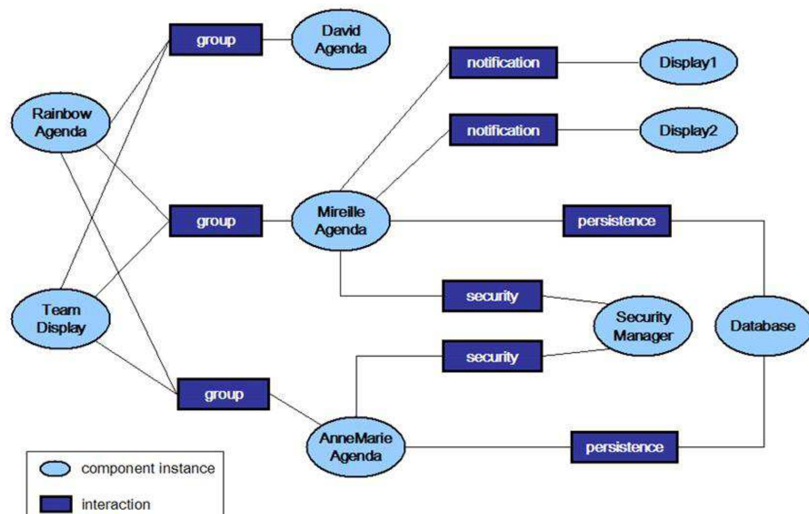



FIGURE 6.1: GRAPHE D'INTERACTIONS POUR L'APPLICATION AGENDA

**Définition d'un schéma d'interactions** L'utilisateur exprime les schémas d'interactions en utilisant le Langage de Spécification des Interactions : ISL (cf. figure 6.2 pour un exemple) qui est décrit plus amplement dans [BFCE+04]. Au travers de ces exemples nous en donnons une définition intuitive. Un schéma décrit plusieurs règles d'interactions qui expriment les contrôles qui doivent être opérés sur les composants liés. Une règle est composée d'une partie gauche qui exprime le modèle de message que l'on veut contrôler et d'une partie droite, l'activité qui correspond à la réécriture du message identifié à gauche de la règle. Un message correspondant au modèle en partie gauche est dit "*notifiant*". Lors de l'instanciation du schéma, il sera identifié relativement aux opérations des composants liés.
Par exemple, le développeur de l'application d'agendas veut autoriser ses utilisateurs à lier à l'exécution un agenda à un " afficheur " afin d'être notifié chaque fois qu'un rendez-vous est ajouté dans l'agenda. Pour les agendas, il peut être utile de gérer la persistance par stockage dans une base de données définie par l'utilisateur. Pour cela, le développeur définit des schémas d'interactions qui sont fournis avec l'application (cf. figure 6.2, schémas *notification* et *persistence*). Ces schémas pourront être instanciés par l'utilisateur final pour lier des composants, nous parlons alors de la pose d'interaction. En cours d'exécution, un utilisateur des agendas peut vouloir lier des agendas, interaction non prévue par le développeur de l'application, pour par exemple créer des agendas de groupe. De la même manière que précédemment, il définit alors son propre schéma d'interactions (cf. figure 6.2, schéma *group*).

**Pose, composition et retrait d'interactions** Lors de l'utilisation de l'application d'agendas, plusieurs schémas d'interactions sont instanciés.

ISL editor

```
interaction notification(Object O, Display display){
   O.* -> O._call // display.notify(_reifiedCall)
}

interaction group(Agenda team, Agenda member, Display display){
   team.addMeeting(string title)
        -> team._call;
           member.addMeeting(title)
   ,
   member.addMeeting(string title)
        -> member._call;
           display.notify(_reifiedCall)
}

interaction persistence(Agenda A, Database database){
   A.addMeeting(string title)
        -> A._call;
           database.store(A.getOwner(),A)
}

interaction security(Object O, SecurityManager service){
   O.* -> if (service.check(_reifiedCall)){
            O._call
          } else {
            throw "UnauthorizedUser"
          }
}
```

**Syntaxe** L'exécution du message notifiant lui-même est notée `_call`. La réification du message peut être passée en argument, elle est notée `_reifiedCall`. L'exécution non ordonnée de deux envois de messages est noté `//`, alors qu'une exécution en séquence est notée `;`. Une synchronisation entre plusieurs activités est notée en étiquetant l'activité par `[n]` et en notant l'attente de fin de cette activité par `_[n]`.

Le schéma *notification* (ligne 1) peut lier deux composants. Il contient une règle (ligne 2). Elle exprime que tout message (utilisation de l'étoile) reçu par un objet ainsi lié, peut être exécuté dans un ordre quelconque avec l'envoi de message au composant *display* associé qui prend en paramètre la réification de l'appel.
Le schéma *group* (ligne 6) peut lier 3 composants. Il définit deux règles. La première stipule que le message *addMeeting* des agendas correspondant au paramètre *team* entraînera après l'exécution du message lui-même (ligne 8), l'ajout du rendez-vous à l'agenda du membre.

FIGURE 6.2: SCHÉMAS D'INTERACTIONS

a) L'instanciation du schéma d'interactions *group* entre les agendas *RainbowAgenda* et *MireilleAgenda* et *TeamDisplay* conduit entre autre à modifier le comportement du composant *MireilleAgenda* et en particulier l'opération *addMeeting*.

b) L'instanciation du schéma d'interactions *security* entre l'agenda *MireilleAgenda* et *SecurityManager* modifie à son tour l'opération *addMeeting*. Une composition des interactions est alors réalisée. Elle a pour conséquence qu'ajouter un rendez-vous dans l'agenda de Mireille correspond à vérifier que cet appel est autorisé et si c'est le cas, à ajouter effectivement le rendez-vous dans l'agenda, puis à notifier l'afficheur. Dans le cas contraire, une exception est levée et l'afficheur n'est pas notifié.

Nous obtenons alors l'interaction suivante sur l'opération *addMeeting* de *MireilleAgenda* :

```
MireilleAgenda.addMeeting(string p0) ->
    if (SecurityManager.check(_reifiedCall)) {
        MireilleAgenda._call ;
        TeamDisplay.notify(_reifiedCall);
    } else {
        throw "UnauthorizedUser"
    }
```

c) Le retrait des interactions correspondant au schéma *group* entre les agendas *RainbowAgenda* et *MireilleAgenda* et *TeamDisplay* (instanciée en a), consiste entre autres à retirer les modifications, décrites dans les règles lignes 7 et 11 de la figure 2, concernant l'opération *addMeeting* du composant *MireilleAgenda*. A présent, ajouter un rendez-vous dans l'agenda de Mireille correspond à vérifier que cet appel est autorisé et si c'est le cas, à ajouter effectivement le rendez-vous dans l'agenda, sans en notifier l'afficheur.

d) L'agenda de *MireilleAgenda* est lié à plusieurs composants (cf. figure 6.1), voici l'interaction résultante :

```
MireilleAgenda.addMeeting(string p0) ->
    Display2.notify(_reifiedCall)
    //  Display1.notify(_reifiedCall)
    //  if(SecurityManager.check(_reifiedCall))
            (MireilleAgenda._call;
            ;   (TeamDisplay.notify(_reifiedCall))
                //  (owner :=(MireilleAgenda.getOwner()
                    ; ComposantBD.store(owner,Database)))
        else {
            throw "UnauthorizedUser"
    }
```

### 6.1.3 Notre approche : une modélisation exécutable des interactions

Pour faciliter une adaptation dynamique des applications à base de composants, nous avons défini un service d'interactions dont nous donnons juste les grandes lignes ici. Sur la base de cette implémentation, nous avons explicité les interactions entre composants

sous la forme de graphe d'activités et défini la composition des interactions sur la base de l'algorithme présenté dans la partie précédente (cf. 3).

#### 6.1.3.1 Service d'interactions

La mise en œuvre des interactions repose sur la définition d'un service d'interactions [BFCE+04]. Une présentation plus détaillée est donnée au paragraphe 6.4.
Les interactions sont décrites dans un langage indépendant du langage de programmation des composants : *ISL (Interaction Specification Language).*

Les schémas d'interactions sont enregistrés dans un serveur spécifique : *le serveur d'interactions.* A partir des schémas, des interactions sont générées à la demande et sont posées sur les composants. Cette approche permet de lier et d'adapter dynamiquement les composants hétérogènes. Les interactions induisent des communications directes entre les composants.
Basée sur le langage ISL, la composition des interactions assure la commutativité et l'associativité de la pose des interactions, ce qui permet une adaptation " cohérente " de l'application par plusieurs intervenants.
Le modèle des interactions et le service d'interactions ont fait l'objet d'un dépôt APP.

#### 6.1.3.2 Des interactions aux compositions d'activités

Nous abordons cette problématique selon le point de vue suivant. A chaque opération d'une instance de composant est associée une modification de comportement, que nous nommons *interaction*. Les interactions sont des activités composites.
Le modèle des activités présenté dans la partie I est ici raffiné pour correspondre à la gestion des interactions entre composants hétérogènes, et les fonctions sur domaine sont précisées dans ce domaine.
Nous distinguons ainsi le raffinement du modèle d'activité pour prendre en compte les interactions et leur composition (cf. 6.2), de l'utilisation de ce modèle pour prendre en compte les règles d'interactions (cf. 6.3).
Nous définissons par la fonction $\bigodot$ l'application d'une interaction à une opération d'un composant[3].
Nous avons les propriétés suivantes :
Soient $i1, ...in$ des interactions, $op$ une opération
$\bigoplus_{interactions}\{i1, i2\} \bigodot op = i1 \bigodot (i2 \bigodot op)$
Dans la section suivante (cf. 6.2) nous définissons la fonction $\bigoplus_{interactions}$ sur une extension de MM4CA, puis nous explicitons la fonction $\bigodot$ dans la section 6.3.

## 6.2 Formalisation et composition dans le domaine des interactions

Le domaine décrit dans cette partie est une extension du domaine présenté comme support d'exemple dans la partie I.

3. Cette notation correspond à l'application d'une modification à une introduction dans [ALB+07]. Nous nous trouvons dans un contexte équivalent même si les éléments ne sont pas de même nature.

### 6.2.1 Domaine applicatif et contraintes sur domaine

La figure 6.3 présente l'extension du métamodèle MM4CA au domaine des interactions.

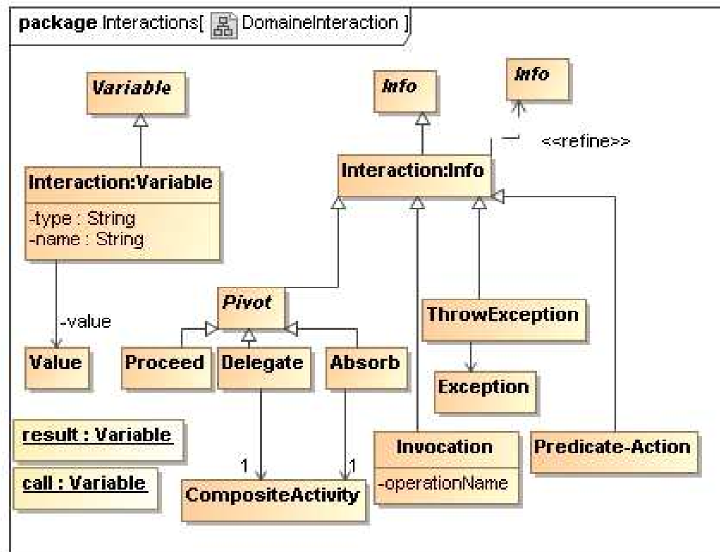



FIGURE 6.3: DOMAINE DES INTERACTIONS

#### 6.2.1.1 Variables

Les variables sont représentées par un nom, un type et une valeur. La valeur nous permet de définir des constantes.

**Equivalence de variables** Deux variables sont équivalentes si elles ont le même type et leur valeurs sont équivalentes.
Deux variables sont pré-définies : *result* qui désigne la valeur de retour de toute interaction, et *call* qui désigne l'appel réifié.

#### 6.2.1.2 Informations

Les informations définissent lever une exception (*throw(exception)*), invoquer une opération (*invocation(operationName)*), vérifier un prédicat (*predicate(predicateName)*) ou correspondent à une information pivot.
Les informations pivots désignent soit le comportement initial (*proceed*), soit une activité composite $f1$ de remplacement (*delegate(f1)*), soit la disparition du comportement initial remplacé par une activité composite $f1$ qui absorbe toutes les activités qui suivent (*absorb(f1)*).

La figure 6.4 visualise au travers d'un exemple la correspondance entre ISL et notre formalisation.
La première variable d'une action d'invocation dénote le destinataire de l'invocation. Lorsqu'au niveau du langage nous trouvons des exceptions, au niveau du modèle nous

| $f_{info}$ | (proceed,null) | (delegate,f1) | (absorb,f1) |
|---|---|---|---|
| (proceed,null) | true | true | true |
| (delegate,f2) | true | $f1 \equiv_\sigma f2$ | true |
| (absorb,f2) | true | true | $f1 \equiv_\sigma f2$ |

TABLE 6.1: EQUIVALENCE DES INFORMATIONS

avons fait le choix d'introduire une activité "absorb". Relativement à la version initiale du langage, elle permet d'obtenir le même comportement mais en offrant plus de flexibilité.

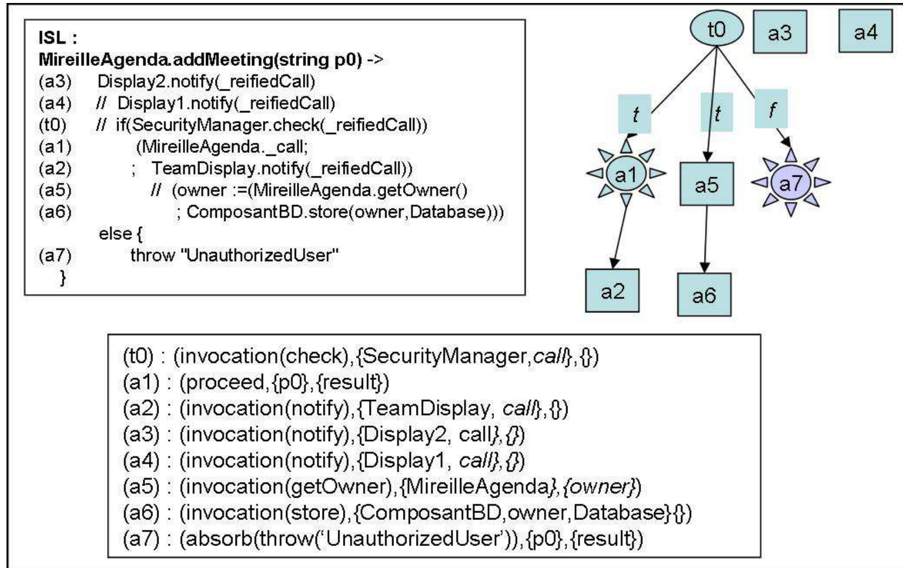



FIGURE 6.4: ISL, INTERACTIONS ET FORMALISATION

**Equivalence d'informations** Deux informations relatives à une activité pivot sont équivalentes relativement au tableau 6.1. Deux informations relatives à un prédicat, une invocation ou une levée d'exception sont équivalentes si les termes sont égaux.

#### 6.2.1.3 Contraintes sur domaine

Nous reprenons ici les contraintes qui avaient été données dans l'exemple fil rouge et les étendons au domaine des interactions.

1. Une seule valeur de retour est acceptée pour toutes les activités.
2. Dans une interaction, il existe au plus une activité correspondant à une information *proceed, delegate ou absorb* par branche (donc ces activités dans une même interaction sont exclusives).
3. Les activités pivots ne peuvent pas prendre en paramètre des constantes, la variable en sortie est toujours *result* ; toutes les variables en entrée des activités pivots dans une même activité composite sont les mêmes.
4. L'activité composite référencée par un *delegate* ou un *absorb* ne contient pas d'activités pivots ; pour l'instant, nous ne cherchons pas à les composer.

5. L'accès en lecture aux variables au sein d'une activité composite n'est possible que si les variables ont précédemment été affectées ou correspondent aux variables d'entrées d'une activité pivot [4].
6. Une activité composite est déterministe relativement à la définition donnée pour la propriété 3.
7. Seules les activités composites référencées par une activité pivot d'information "absorb" contiennent des d'activités correspondant à la levée d'une exception.
8. Une activité composite contient au moins une activité pivot.

### 6.2.2 Détection des ensembles de fusion

La détection des paires de fusion vise à fusionner les activités caractérisant le comportement que l'on veut enrichir ou modifier.

#### 6.2.2.1 Activités pivots

Les activités pivots correspondent aux informations, dites pivots : *proceed, delegate* et *absorb*.

#### 6.2.2.2 Détection des paires de fusion

Deux pivots sont potentiellement fusionnables s'ils ne sont pas assujettis à des conditions qui s'excluent. Dans tous les autres cas, si la fusion n'est pas possible, pour les raisons qui suivent, une erreur *conflit de pivots* est levée.

**erreur de signatures** : les schémas d'interactions ne sont fusionnables que si les parties gauches désignent la même activité, donc que le nombre de variables des activités pivots sont les mêmes. Comme nous n'autorisons pas l'utilisation de constantes dans ces activités, l'unification est toujours possible lorsque le nombre de paramètres est le même et qu'il y a égalité des types respectifs des variables.

**erreur de fusion des contrôles** : selon les contrôles opérés sur les activités, il n'est pas possible de les fusionner. Ainsi que signifie : absorber un comportement et en même temps le déléguer, déléguer un comportement de deux manières, ... Ces cas sont notés comme non définis dans le tableau 6.2, et en conséquence déclenchent une erreur. Nous explicitons ces choix à présent.

La détection de deux activités *delegate* ou *absorb* potentiellement fusionnables déclenche une erreur parce qu'offrir deux comportements en lieu et place d'un comportement initial nécessiterait d'expliciter une composition, qui aujourd'hui ne nous est pas apparue comme évidente. Nous avons choisi à ce jour de ne pas considérer l'équivalence des activités composites pour accepter certaines fusions ; ce point semble une extension facile de notre travail.
Le refus de composer une activité *absorb* et *delegate* est lui discutable, puisque intuitivement, l'absorbtion devrait avoir la priorité. Néanmoins en acceptant cette fusion nous perdrions l'associativité de la composition (cf. la vérification de la propriété d'associativité dans le domaine des interactions, page 95).

4. En effet, ces variables désignent les paramètres de l'appel, ont donc une valeur à l'exécution.

| $f_{info}$ | (proceed,null) | (delegate,f1) | (absorb,f1) |
|---|---|---|---|
| (proceed,null) | (proceed,null) | (delegate,f1) | (absorb,f1) |
| (delegate,f2) | (delegate,f2) | non définie | non définie |
| (absorb,f2) | (absorb,f2) | non définie | non définie |

TABLE 6.2: COMPOSITION DES INFORMATIONS : $mergeInfo$

#### 6.2.2.3 Propriétés

**Eléments d'un ensemble de fusion** L'extension du domaine Fil rouge ne modifie pas la propriété FR1 : les éléments des ensembles de fusion correspondent à des activités issues d'activités composites toutes différentes et l'union des gardes portant sur les activités d'un ensemble de fusion est exclusive vis-à-vis de tout autre ensemble de fusion.

**Cardinalités et complexité** Nous nous trouvons dans le même cas que dans le domaine fil rouge. En conséquence le nombre de paires de fusion est défini par :
Soient $ca_1...ca_n, n$ activités composites composées chacune de $p_i$ pivots (donc branches).
Le nombre maximum de paires de fusion sera de : $\sum_{i=1}^{i=n-1} \sum_{j=i+1}^{j=n} p_i * p_j$.
Le nombre de cliques maximales sera lui borné par : $p_1 * p_2 * ... * p_n$
La cardinalité maximale d'un ensemble de fusion est de n.
La complexité du calcul des paires est donc réduite par le fait que nous ne comparons pas toutes les activités pivots deux à deux mais seulement les activités pivots appartenant à des activités composites différentes. De plus dans ce domaine le nombre des activités pivots est restreint puisqu'il est proportionnel aux nombres de branches dans une activité composite.

**Respect de la propriété 6 sur domaine relative à l'absence de paires de fusion au sein d'une activité composite.** Cette propriété est respectée par les contraintes qui définissent le domaine des interactions : une seule activité pivots par branche de l'interaction, donc deux activités pivots au sein d'une interaction sont forcément exclusives et ne forment donc pas une paire.

**Respect de la propriété 20 sur domaine relative à l'ordre** Le calcul des paires de fusion ne dépend pas de l'ordre des activités par construction. En cas de conflit de pivot, le résultat est une erreur indépendante des calculs antérieurs.

### 6.2.3 Transformations et composition de transformations

Nous utilisons uniquement les transformations définies au niveau du métamodèle MM4CA : $remove$ et $substituteIn$. La composition des substitutions de variables consiste à rechercher l'unificateur plus général.

#### 6.2.3.1 Propriétés de composition des transformations

**Respect de la propriété 1 sur la composition des transformations indépendante de l'ordre :** L'application des transformations de retraits donne un résultat identique

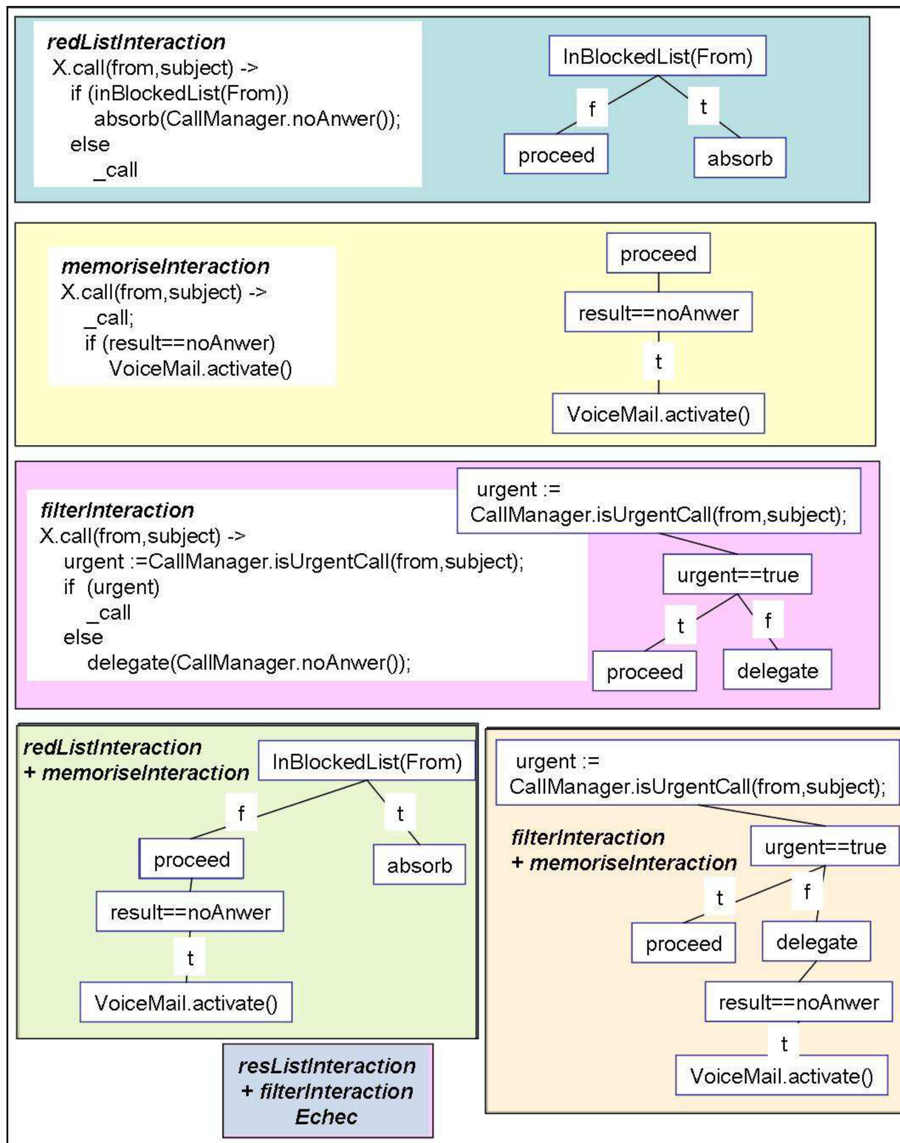
quelque soit l'ordre des retraits. La composition des substitutions repose sur la construction d'un unificateur. Les propriétés suivantes s'appliquent donc. (i) Il y a unicité de unificateur principal au renommage près des variables et la relation d'équivalence prend en compte cette caractéristique. (ii) Nous sommes certain de trouver un unificateur car les cas d'échec de l'algorithme d'unification ne s'appliquent pas : (i) nous ne manipulons pas des termes mais uniquement des variables (pas d'utilisation de la règle de décomposition et donc de conflit), (ii) les variables en partie droite n'apparaissent jamais en partie gauche dans les éléments de substitutions (test de cyclicité inutile).

### 6.2.4 Fusion d'activités simples

La fusion des ensembles de fusion porte sur des activités pivots qui stipulent soit une préservation du comportement initial (*proceed*), soit une redéfinition de ce comportement (*delegate*), soit une absorbtion de ce comportement (*absorb*). Dans ce dernier cas, le comportement initial n'étant pas préservé, les successeurs de cette activité sont également absorbés.
La fonction $mergeInfo$ explicite la fusion des informations (cf. tableau 6.2). Elle est associative ce qui nous permet de la définir sur un ensemble sur la base des données du tableau.
La figure 6.5 visualise des exemples de fusion. Les figures 6.6, 6.7 et 6.8 visualisent différentes compositions en fonction des activités pivots.
La fusion des interactions pouvant "absorber" des activités dans certaines branches, nous faisons le choix de copier les activités successeurs lorsqu'elles sont préservées par la fusion. Cela nous conduit à définir la fonction *duplicate* par analogie à la copie d'activités composites : $duplicate(ActivitySet, R_p, R_c) = (newActivities, newR_p, newR_c)$.
Elle recopie les activités passées en paramètre et les relations qui leur correspondent en tenant compte des nouvelles activités.

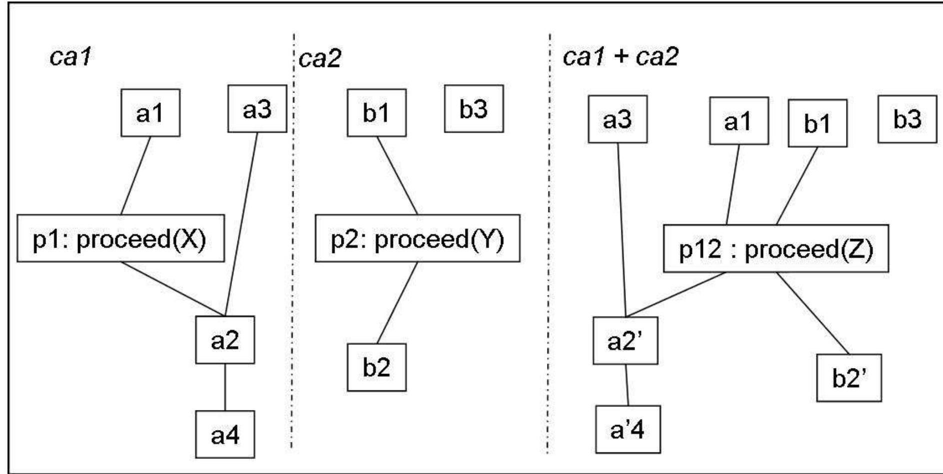
La fusion d'un ensemble d'activités pivots $\{p_1..p_n\}$ telles que $p_i = (id_i, info_i, inputs_i, result)$, $inputs_i = (i_i^1..i_i^k)$ est définie comme :
$mergeSimpleActivities_{Interactions}(\{p_1..p_n\}, R_p, R_c, Al) = (newActivities, newR_p, newR_c, todo_R)$
avec :
$newActivities = \{np\} \cup N_a$
$newR_p = R_{p_{succ}} \cup R_{p_{pivot}}$
$newR_c = R_{csucc} \cup R_{cpivot}$
$todo_R = T_s \cup T_r \cup T_{r_s}$
où les différents ensembles sont construits comme suit :

1. création d'une nouvelle activité $np = (id, info, \{nv^1, .., nv^k\}, \{result\})$ où
   - l'information résulte de la fusion des informations des activités composant l'ensemble de fusion (cf. tableau 6.2) :
     $info = mergeInfo(\{info_1, ..info_n\})$ ;
   - les variables sont déterminées par unification des variables des activités de l'ensemble conduisant à ajouter les transformations de substitutions correspondantes :
     $T_s = \{substituteIn_{(i_i^j, nv^j)}, \forall i \in 1..n, \forall i_i^j \in inputs_i\}$.
2. la nouvelle activité a pour prédécesseur l'ensemble des prédécesseurs des activités pivots :
   $R_{p_{pivot}} = \bigcup_{p_i \in \{p_1..p_n\}} \{(p_j, np) | p_j \in pred(R_p, p_i)\}$



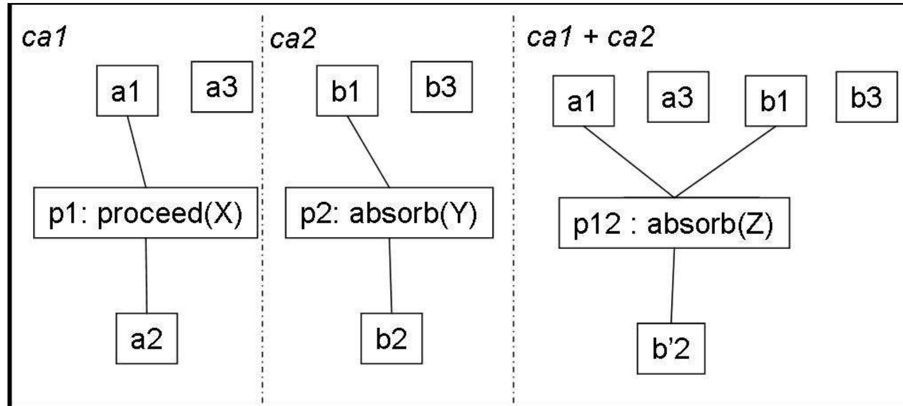

L'interaction (*RedListInteraction*) bloque les appels considérés comme indésirables.
L'interaction (*memoriseInteraction*) mémorise les messages dès qu'un appel n'a pas pu aboutir.
L'interaction (*filterInteraction*) ne laisse passer un appel que si l'appel est considéré comme urgent, sinon aucune réponse n'est donnée.
Les 3 graphes en bas visualisent le résultat de la composition de ces interactions deux à deux. La fusion des interactions *filterInteraction* et *RedListInteraction* échoue parce que nous ne savons pas composer *absorb* et *delegate* lorsque l'appel n'est pas urgent et sur liste rouge.

FIGURE 6.5: EXEMPLES DE COMPOSITION D'INTERACTIONS




Pour simplifier la notation, nous nous autorisons à noter une activité par :
*id : info(variables d'entrée).*
$newActivities = N_a \cup p12, N_a = \{a2', b2', a'4\}$
$newR_p = R_{p_{pivot}} \cup R_{succ},$
$R_{p_{pivot}} = \{(a1, p12), (b1, p12)\}, R_{succ} = \{(p12, a'2), (p12, b'2), (a'2, a'4), (a3, a'2)\}$
$todo_R = T_s \cup T_r \cup T_{r_s},$
$T_s = \{substituteIn_{(X,Z)}, substituteIn_{(Y,Z)}\},$
$T_r = \{remove_{p1}, remove_{p2}\}, T_{r_s} = \{remove_{a2}, remove_{b2}\}$

FIGURE 6.6: EXEMPLE DE COMPOSITION D'ACTIVITÉS AUTOUR DE *proceed*




$newActivities = N_a \cup p12, N_a = \{b2'\}$
$newR_p = R_{p_{pivot}} \cup R_{succ},$
$R_{p_{pivot}} = \{(a1, p12), (b1, p12)\}, R_{succ} = \{(p12, b'2)\}$
$todo_R = T_s \cup T_r \cup T_{r_s},$
$T_s = \{substituteIn_{(X,Z)}, substituteIn_{(Y,Z)}\},$
$T_r = \{remove_{p1}, remove_{p2}\}, T_{r_s} = \{remove_{a2}, remove_{b2}\}$

FIGURE 6.7: EXEMPLE DE COMPOSITION D'ACTIVITÉS AUTOUR DE *absorb*

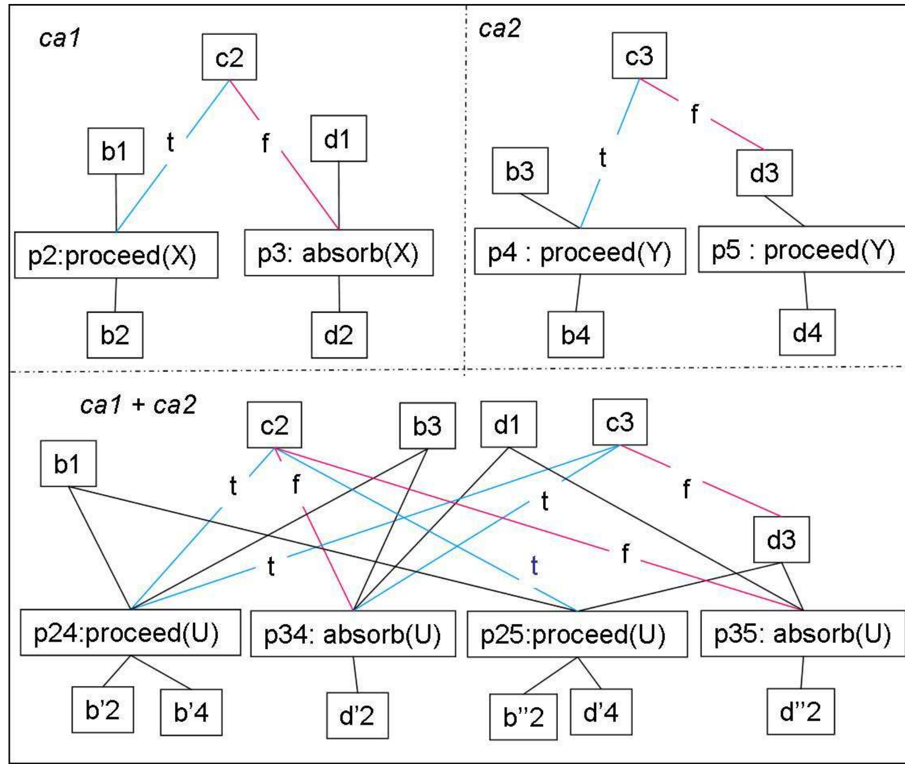



FIGURE 6.8: COMPOSITION D'ACTIVITÉS "PIVOTS" ET CONDITIONNELLES

3. la nouvelle activité a pour conditions l'ensemble des conditions des activités pivots :
   $R_{cpivot} = \bigcup_{p_i \in \{p_1..p_n\}} \{(c_j, np, true) | c_j \in condTrue(R_c, p_i)\} \cup$
   $\bigcup_{p_i \in \{p_1..p_n\}} \{(c_j, np, false) | c_j \in condFalse(R_c, p_i)\}$
4. les activités pivots de l'espace de fusion sont retirées puisque la fusion a créé l'activité de remplacement $np$ :
   $T_r = \{remove_{(p_1)}, ....remove_{(p_n)}\}$
5. les successeurs des activités fusionnées sont soit dupliqués soit absorbés. Dans tous les cas ils seront donc retirés :
   $T_{r_s} = \{remove_{a_j}, \forall a_j \in \bigcup_{p_i \in \{p_1..p_n\}} succ^*(R_p, p_i)\}$ ;
6. les successeurs de la nouvelle activité sont la copie de tout ou partie des activités successeurs des activités pivots selon la composition suivante :
   (a) si la fusion concerne une activité *absorb* notée $p_k$, seuls les successeurs de l'activité $p_k$ deviennent les successeurs de la nouvelle activité ; ils sont dupliqués.
   $Succ = succ^*(R_p, p_k)$
   $R_{p_s} = \{(np, s), \forall s \in succ(R_p, p_k)\}$
   $(Na, R_{p_{succ}}, R_{csucc}) = duplicate(Succ, R_p \cup R_{p_s}, R_c)$
   (b) dans le cas contraire (aucune activité *absorb* dans l'ensemble de fusion), tous les successeurs des activités formant l'espace de fusion sont les sucesseurs de la nouvelle activité ; ils sont dupliqués [5] :
   $Succ = \bigcup_{i=1}^{i=n} succ^*(R_p, p_i)$

5. La fusion avec un *aborb* peut absorber les activités, qui seront donc retirées, c'est pourquoi, dans ce domaine, contrairement au domaine fil rouge, il est essentiel de dupliquer les activités.

$R_{p_s} = \{(np, s), \forall s \in \bigcup_{p_i \in \{p_1..p_n\}} succ(R_p, p_i)\}$
$(Na, R_{p_{succ}}, R_{csucc}) = duplicate(Succ, R_p \cup R_{p_s}, R_c)$

#### 6.2.4.1 Propriétés de la fusion d'activités simples

**respect de la propriété 3 sur domaine relative à la non-création de circuit** Pour qu'il y ait circuit, il faudrait qu'après l'application des transformations, une activité ait comme successeur une activité qui la précède ou conditionne son exécution. Or par fusion, les nouvelles activités ont pour prédécesseur l'union des prédécesseurs et des conditions des activités fusionnées. Celles-ci appartiennent forcément ou à des activité composites différentes (une seule itération du processus de composition) des successeurs ou à une même activité composite qui par hypothèse ne contient pas de circuit.

**respect de la propriété 4 sur domaine relative à l'absence d'incohérence** Seules des activités non exclusives sont fusionnées et la seule opération de modification des relations de condition consiste à faire l'union de conditions non exclusives. Les successeurs des activités pivots ayant été dupliqués ils sont conditionnés directement par des conditions ne s'excluent pas et par transitivité par les conditions de la nouvelle activité pivot qui ne s'excluent pas.

**respect de la propriété 2 sur domaine relative à l'indépendance de l'ordre des activités** Cette propriété se démontre par l'indépendance de l'ordre dans la composition des transformations et le fait que chaque fusion d'activités simples traite de manière équivalente toutes les activités et impose l'unicité des activités *absorb*.

### 6.2.5 Normalisation partielle

Nous avons démontré que la fusion engendre des ensembles d'activités qui respectent les propriétés de validité. En composant toutes les transformations, ces propriétés restent vraies parce que les seules transformations qui agissent sur des activités partagées sont celles relatives à la substitution des variables. En effet les retraits d'activités ne peuvent pas invalider les ensembles d'activités, et les modifications de relations interviennent uniquement sur de nouvelles activités. Les substitutions de variables interviennent uniquement sur des variables libres et en conséquence ne peuvent pas engendrer d'erreur.
La normalisation partielle est donc inutile et correspond donc à l'identité.

### 6.2.6 Normalisation sur domaine

La fonction $normaliseOnDomain_{Interactions}$ vérifie que les contraintes énoncées par le domaine (cf. 6.2.1.3) sont vérifiées.
Cette fonction dans ce domaine vérifie seulement si les contraintes sont vérifiées et lève une erreur *impossible normalisation* dans le cas contraire.
Par construction, les contraintes de *1 à 5* et *7 et 8* sont vérifiées sur les ensembles d'activités et de relations résultant des compositions. La contrainte *6* relative au déterminisme n'est par contre pas forcément vérifiée puisque par fusion une activité peut avoir plusieurs activités prédécesseurs partageant une même variable. La normalisation sur domaine ne résout pas le problème. Une erreur *impossible normalisation* est levée.

### 6.2.7 Composition d'un ensemble d'activités composites

#### 6.2.7.1 Traitements des ensembles de fusion

Tous les ensembles de fusion sont traités, dans n'importe quel ordre. Si une erreur intervient, l'ensemble de la composition est annulé.

#### 6.2.7.2 Pas de réintroduction de paires de fusion

Le respect de la propriété 6 sur le domaine des interactions (cf. 6.2.2.3) nous permet d'assurer que la composition d'activités composites dans le domaine des interactions est idempotente (cf. propriété 5).
Par contre les seules interactions qui sont idempotentes sont celles qui sont composées d'une unique activité *proceed*.

#### 6.2.7.3 Respect des propriétés exigées

**Décroissance du nombre d'ensembles de fusion** La fusion d'un ensemble de fusion engendre des activités qui ne formeront plus un ensemble de fusion. En effet, la fusion d'un ensemble de fusion génère une nouvelle activité pivot qui est assujettie à l'union des gardes des activités fusionnées ; or nous avons démontré (cf. 6.2.2.3) que les ensembles de fusion s'excluent.

**Validité des activités** Nous avons montré que la composition des interactions procède en un seul tour, ne créé pas de circuits et engendre des activités valides.

#### 6.2.7.4 Remarque

Nous n'avons pas forcé à ce qu'une activité composite exprime tous les cas possibles, c-à-d. que pour toute activité pivot conditionnée pas $c1$ il existe une autre activité pivot conditionnée par $nonc1$ dans l'activité composite. La présence d'une branche contenant une activité pivot conditionnée par $c1$ et l'absence d'une branche $nonc1$ contenant également une activité pivot signifie "ne rien faire". Lors des compositions c'est le comportement obtenu. Nous ne trouverons pas d'activité pivot dans une branche où $c1$ ne serait pas vérifiée.

### 6.2.8 Propriétés des compositions d'interactions

#### 6.2.8.1 Idempotence

En composant plusieurs interactions, les seules interactions qui seront idempotentes, sont celles réduites à un *proceed*. Dans tous les autres cas, ou bien il y aura échec de composition (absorb avec absorb ou delegate avec delegate), ou bien les activités associées au *proceed* seront dupliquées.

#### 6.2.8.2 Préservation "partielle" des propriétés

**Il n'y a pas préservation des activités.** La présence d'une activité *absorb* viole la préservation des activités puisqu'il y a disparition des branches absorbées. Si nous éliminons cette sorte d'activité, la préservation des activités est vérifiée, modulo la fonction d'équivalence que nous avons défini (cf. tableau 6.1).

$\Longrightarrow$En conclusion la *préservation des activités* n'est potentiellement pas vérifiée sur la composition d'interactions en présence d'activités *absorb*.

**Il y a préservation des relations.** La préservation des relations de précédence et de conditions est vérifiée. En effet toutes les activités préservées ont au moins les mêmes précédents et successeurs à l'équivalence près, et sont assujetties aux mêmes gardes. Lors d'une fusion de *proceed* ou *delegate*, l'activité résultante de la fusion a pour prédécesseurs (resp. successeurs) l'union des prédécesseurs (resp. successeurs). Les conditions portant sur la nouvelle activité sont l'union des conditions de toutes les activités fusionnées. Lors de l'absorption d'activités, les activités non absorbées ne voient pas les relations qui les concernent être modifiées. L'activité *absorb* est par contre assujettie à l'union des conditions des activités absorbées et donc les conditions et précédences qu'elles devaient respecter ne sont pas modifiées.
$\Longrightarrow$ Ainsi la composition d'interactions garantit que l'ordre entre les activités issues de composition est préservé, mais, par absorption, des activités peuvent disparaître.

#### 6.2.8.3 Non Duplication par la fonction de composition des interactions

Si nous prenons comme hypothèse qu'il n'existe pas d'activités équivalentes, à l'exclusion des activités pivots, dans les interactions à composer, nous démontrons que la composition d'interactions n'engendre pas d'activités dupliquées.
Dans la composition des interactions, les activités successeurs des activités pivots sont dupliquées. Si une activité pivot (*proceed* ou *delegate*) intervient dans un seul ensemble de fusion les activités de la branche sont recopiées une seule fois et les activités correspondantes sont retirées, il n'y a donc pas de "duplication". Si une activité pivot intervient dans $n$ ensembles de fusion, ses successeurs sont recopiés n fois. Cependant puisque l'activité pivot appartient à n ensembles, cela implique que les conditions s'excluent, sinon elles auraient appartenues au même ensemble. Donc les activités ainsi recopiées s'excluent et ne sont pas équivalentes.

Par contre si parmi les activités des interactions initiales, certaines sont équivalentes sans être dupliquées, la composition peut les dupliquer. Supposons que deux interactions définissent comme successeur de l'activité pivot *proceed*, une activité équivalente $a$, mais l'une étant conditionnée à $c$ vraie et l'autre non conditionnée, le résultat de la composition engendrera deux activités $a$ dupliquées parce que équivalentes et soumises à la même condition. Néanmoins nous pouvons considérer que la fonction de composition ne duplique pas les activités puisque les activités dupliquées étaient initialement équivalentes (cf. définition de l'absence de duplication page 66).

#### 6.2.8.4 Indépendance des ordres de composition

Nous avons démontré l'indépendance de l'ordre des activités pour calculer les paires de fusion (cf. page 90) et la fusion (cf. page 92). Nous avons également démontré l'indépendance de l'application des transformations vis-à-vis de l'ordre des transformations (cf. page 90). Ces différents points nous permettent d'assurer l'indépendance de l'ordre des activités composites.

| $f_{info}$ | proceed | delegate | absorb |
|---|---|---|---|
| (absorb,proceed)=absorb | absorb | erreur | erreur |
| (absorb \| delegate,absorb \| delegate)=erreur | - | - | - |
| (proceed,proceed)=proceed | proceed | delegate | absorb |
| (proceed,delegate)=delegate | delegate | erreur | erreur |

TABLE 6.3: FUSION DES ACTIVITÉS PIVOTS ET ASSOCIATIVITÉ

#### 6.2.8.5 Associativité

Des interactions sont ajoutées tout au long du cycle de vie des composants. Il est donc important de pouvoir assurer le développeur que la composition des interactions ne dépend pas des temps où elles ont été ajoutées, sans quoi nous ne saurions garantir une réelle indépendance des préoccupations.
Le tableau 6.3 visualise les compositions ternaires. Nous constatons sur ce tableau que la fusion des seules activités pivot est bien associative, il nous faut donc démontrer que les transformations et les relations et activités ajoutées sont bien équivalentes.
Nous commençons par éliminer les cas d'erreurs. Les erreurs de composition peuvent être provoquées par :
- les activités pivots n'ont pas la même arité *(échec de* $detectMergePairs$*)* ;
- les activités pivots ne sont pas fusionnables *(échec de* $detectMergePairs$*)*
- l'activité résultante n'est pas déterministe *(échec de* $normalizeOnDomain$*)*
La fusion des activités pivots ne modifiant pas l'arité des pivots, si la première erreur est levée, lors de la composition de 2 activités composites, l'erreur se produira quelques soient les compositions précédentes. Le tableau 6.3 montre que les erreurs se produisent quelques soient les ordres de composition. Enfin le non-déterminisme de l'activité résultante suppose que dans le résultat d'une composition nous avons trouvé deux activités qui accèdent à une même variable dans un ordre non déterminé au moins une fois en écriture. La composition dans le domaine des interactions ne garantissant pas la préservation des activités, l'erreur pourra ne pas se reproduire. Nous n'avons donc pas l'associativité en cas d'erreur de composition dans le cas d'une détection de non-déterminisme et d'une activité pivot "absorb". La normalisation sur domaine n'ayant aucun impact sur la composition autre que de lever une erreur elle n'intervient pas davantage dans la démonstration.
Nous avons démontré dans le cadre de l'exemple fil rouge (cf. page 70) que la composition des activités seulement composées d'activités pivots *proceed* est associative. Le comportement de la fusion, s'il n'y a pas d'erreur, en prenant en compte les activités *delegate* étant le même, nous avons également l'associativité en prenant en compte les activités *delegate*[6].
La fusion d'activités mettant en jeu une activité *absorb* est par contre différente. Une seule activité *absorb* est possible dans une composition, sinon il y a erreur.
Soient $(p1, p2, p3)$ des pivots respectifs de $(ca1, ca2, ca3)$ tels que $p1$ et $p2$ sont des activités *proceed* et $p3$ est une activité *absorb*, seul cas de composition ternaire qui ne crée pas d'erreur. Quels que soient les ensembles de fusion, la fusion procède par unification des variables qui est, elle, associative. En cas d'activités *absorb* dans un ensemble de fusion,

6. La fusion procède dans le domaine des interactions par copie des successeurs, alors que dans le domaine Fil rouge, qui n'a pas l'activité absorbante, nous laissons la normalisation partielle gérer la validité. Les comportements sont néanmoins équivalents au regard de l'associativité

l'union des successeurs des activités absorbés sont retirés, pour les prédécesseurs et les conditions nous procédons par l'union qui est associative. Nous nous intéressons donc uniquement aux successeurs. La préservation des propriétés nous assure que par composition tous les successeurs des activités sont toujours présents s'il n'y a pas d'activité *absorb*. Donc après la composition de $ca1$ et $ca2$, nous savons que les successeurs de $p12$, activité *proceed*, résultant de la fusion de $p1$ et $p2$ sont l'union des successeurs de $p1$ et $p2$. La composition de $p3$ avec $p12$ absorbera donc l'union des successeurs de $p1$ et $p2$, et l'activité résultante $p123$ est une activité *absorb*. Si à présent nous composons $p1$ et $p3$, la nouvelle activité $p13$ aura comme seuls successeurs ceux de $p3$. Si nous composons $p13$ avec $p2$, il y aura absorbtion des successeurs de $p2$, et la nouvelle activité sera alors bien équivalente à $p123$.

## 6.3 Formalisation des schémas et des règles d'interactions

Nous explicitons à présent comment passer des activités composites aux schémas d'interactions.

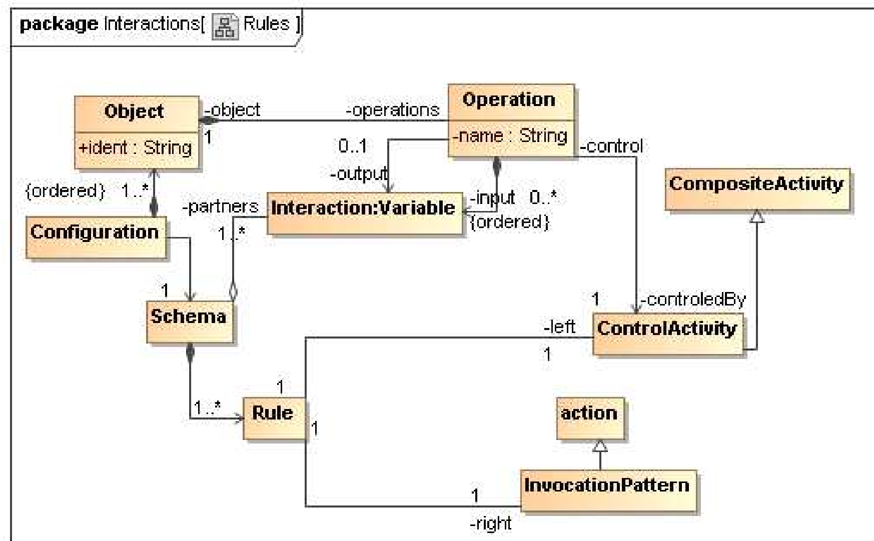



FIGURE 6.9: SCHÉMAS, RÈGLES D'INTERACTIONS ET ACTIVITÉS COMPOSITES

Les objets sont des représentants au niveau du modèle [7] des instances de classes ou de composants qui sont sujets à des adaptations par pose d'interactions.
De même les opérations sont des représentants des opérations des composants qui sont contrôlées par interactions.

### 6.3.1 Modélisation

#### 6.3.1.1 Objets et Opérations

*Objet*

7. et nous verrons au chapitre suivant également au niveau du service d'interactions

Un *objet* $o = (id_o, ops)$ est défini par un identifiant unique dans le système $id_o$ et une liste d'opérations $ops$.
Nous notons $\mathcal{O}$, l'ensemble des objets.

*Opération*

Une *opération* est définie par un tuple $(id_o, opName, inputs, ca)$ où $id_o$ est l'identifiant de l'objet auquel elle est associée, $opName$ est le nom de l'opération, $inputs$ la liste de variables d'entrée, et $ca$ est une activité composite qui explicite les modifications du comportement de l'opération.

Initialement à toute opération correspond une activité composite constituée d'une seule activité dont l'action est *proceed*, et dont les variables correspondent aux variables de l'opération. Nous utilisons l'idempotence de cette activité.

Nous notons $\mathcal{OP}$, l'ensemble des opérations.

Les activités de modification du comportement jouent le même rôle que des wrappers [ALB+07], mais présentent la propriété d'être composées et non pas encapsulées les unes dans les autres.

**Exemple**

Soit l'objet : $(DocAgenda, \{addMeeting_{Doc}\})$

Soit l'opération : $addMeeting_{Doc} = (DocAgenda, addMeeting, \{intitule\}, c_{addMeeting_{Doc}})$ où
$c_{addMeeting_{Doc}} = (\{proc_{addMeeting_{Doc}}\}, \{\}, \{\})$, avec
$proc_{addMeeting_{Doc}} \rightarrow (proceed, \{intitule\}, result)$

#### 6.3.1.2 Règles et Interactions

*Règle*

Une *règle* est définie par un couple : (*InvocationPattern*,*ControlActivity*)

Un *InvocationPattern* est un couple $(receiver, action)$ où
- $receiver$ est une variable qui désigne l'objet dont l'opération doit être contrôlée,
- $action$ est une action qui a pour information un terme de la forme : *invocation(OperationName)* où *OperationName* est une variable ou un atome. L'unique variable de sortie de l'action est result.

Une *ControlActivity* est une activité composite dans laquelle toutes les activités pivots ont pour variables d'entrées, les variables d'entrées de l'*invocationPattern* de la règle. Dans une activité de contrôle certaines variables sont utilisées uniquement en lecture et ne sont donc pas initialisées au sein de l'activité composite. Elles désignent les *partenaires* de la règle.
Nous associons à la définition d'une règle $r = (ip, ca)$, les fonctions suivantes :
-$invocationPattern(r) = ip, ip = (o, (invocation(Op), inputs, output)), inputs = \{v_1..v_n\}$
-$controlActivity(r) = ca$
-$receiver(r) = receiver(ip) = o$
-$inputs(r) = inputs(ip)$

-$partners(r) = (inputs(ca) \cup \{receiver(r)\}) \backslash (inputs(ip) \cup \{call\})$

Nous notons $\mathcal{R}$, l'ensemble des règles.

**Remarques :**
- La variable spéciale *call*, qui désigne l'appel réifié, et la variable *result*, ne sont pas renommées par copie des activités.
- Nous ne pouvons pas aujourd'hui exprimer de contraintes sur les paramètres des opérations contrôlées. Nous pensons lever cette limitation en construisant des partitions de l'espace d'invocation à la manière du travail réalisé dans la thèse de Olivier Nano [Nan04].

*Interaction*

Lorsque dans une règle, le receveur, le nom de l'opération, les partenaires correspondent tous à des constantes, nous appelons cette règle une interaction.
A une interaction $i$ correspond donc une unique opération que nous notons :$operation(i)$.
Nous notons $\mathcal{I}$, l'ensemble des interactions, $\mathcal{I} \subseteq \mathcal{R}$.

#### 6.3.1.3 Schémas

*Schéma*

Un schéma est défini par un couple *(partners,rules)* où *partners* est une liste de variables désignant les partenaires potentiels du schéma ($card(partners) \geq 1$) et *rules* est un ensemble de règles.

**Propriété 6.1:** *Validité d'un schéma*

Un schéma $s$ est valide [8] si :
- toutes les règles de $s$ ont pour receveur un partenaire de $s$
  $\forall r \in rules(s), receiver(r) \in partners(s)$
- tous les partenaires des règles sont inclus dans l'ensemble des partenaires de $s$
  $\forall r \in rules(s), partners(r) \subseteq partners(s)$

**Exemple**

Soit un schéma *notification* composé d'une règle décrite ici dans la syntaxe ISL :

```
interaction notification(Agenda a, Display display){
    a.addMeeting(string title) -> o._call //
        display.notify(_reifiedCall)
}
```

Ce schéma exprime qu'en cas d'ajout d'un rendez-vous à l'agenda partenaire représenté par la variable $a$, l'afficheur partenaire représenté par la variable $display$ doit être notifié ; ces 2 activités s'exécutent dans un ordre non déterminé.

Il sera formalisé comme :
$notification(\{a, display\}, \{r1\})$ $r1 = (ip, ca)$,
$ip = (a, (invocation(addMeeting), \{title\}, \{result\}))$
$ca = (\{id_1, id_2\}, \{\}, \{\})$, avec
$id_1 \rightarrow (proceed, \{title\}, \{result\}), id_2 \rightarrow invocation(invocation(notify), \{display, call\}, \{\})$
Nous avons donc :
-$receiver(r1) = a$

8. Nous utilisons l'égalité des variables pour évaluer cette propriété

-$inputs(r1) = \{title\}$
-$partners(r1) = \{a, display\}$

Nous notons $\mathcal{S}$ l'ensemble des schémas valides.

#### 6.3.1.4 Configuration

*Configuration*
Une configuration est un couple $(\sigma, s)$ constitué d'une substitution et d'un schéma tel que la substitution $\sigma = \{(v_1, c_1), ..(v_n, c_n)\}$ associe par des constantes [9] les partenaires au schéma :
$\forall v_i \in parters(s), \exists (v_i, c_i) \in \sigma$

**Exemple**
Soit la configuration [10] $conf = (\{(a, DocAgenda), (display, TeamDisplay)\}, notification)$

**Propriété 6.2:** *Variables compatibles*
Nous considérons que deux tuples de variables $l1$ et $l2$ sont compatibles si : $card(l1) = card(l2)$ et $\forall l1_i \in l1, type(l1_i) = type(l2_i)$, nous notons $l1 \cong l2$ [11]
**Propriété 6.3:** *Configuration bien construite*
Une configuration $(\{(v_1, p_1)..(v_n, p_n)\}, s)$ est bien construite si :
- les patterns d'invocation des règles s'appliquent aux partenaires :
  $\forall r \in rules(s)$, soit $invocationPattern(r) = (v_j, invocation(opName), inputs, \{result\})$
  $\exists op \in operations(p_j)$, telle que $name(op) = opName, inputs(op) \cong inputs$
- les activités de contrôles peuvent s'appliquer aux partenaires :
  $\forall r \in rules(s), \forall a \in activities(controlActivity(r))$ telles que
  $info(a) = invocation(opName)$ et $receiver(a) = v_j \in partners(r)$,
  alors $\exists op_i \in operations(p_j)$ telle que $name(op_i) = opName$& $inputs(op_i) \cong inputs(a)$

Nous notons $\mathcal{C}$ l'ensemble des configurations bien construites.

### 6.3.2 Pose et compositions d'interactions

A partir d'une configuration, nous construisons les interactions correspondantes (règles instanciées) puis les appliquons aux opérations mises en jeu par la configuration.

**Instanciation d'une règle** En fonction d'une configuration, une règle peut s'appliquer sur plusieurs opérations d'un même objet. "Instancier" une règle en fonction d'une configuration revient donc à calculer des "***interactions***" dont le receveur et le nom de l'opération de l'*invocationPattern* ont été instanciés, c-à-d. sont à présent des constantes.
**Fonction 7** (F7) : *Instanciation d'une règle*
$instanciate$ est définie de : $\mathcal{C} \times \mathcal{R} \longrightarrow \mathcal{I}^n$
Soient $c = (\sigma_c, s), r = (v_i, (invocation(Name), inputs, \{result\}), ca) \in rules(s)$,
$(v_i, c_i) \in \sigma_c, o_i = c_i.value$,

9. variable dont la valeur est l'identifiant d'un objet.
10. Pour des raisons de concision, nous confondons ici la constante et sa valeur.
11. Nous faisons le choix d'une égalité de type dans cette formalisation qui se focalise sur la composition des activités. Cependant les implémentations prennent en compte le sous-typage et la thèse de Audrey Occello [Occ06] établit formellement ces vérifications, tandis que dans sa thèse Olivier Nano propose un partitionnement des signatures[Nan04].

$instanciate(c, r) = \{i_1, ..i_n\}$, avec
$\forall op_i = (o_i, Name_i, inputs_i, a_i) \in operations(o_i)$,
$inputs_i \cong inputs, Name \equiv_\sigma Name_i$ [12]
$r_i = ((\sigma_c(v_i), invocation(Name_i), inputs, \sigma_c(ca)))$

**Pose d'interactions** Plusieurs interactions peuvent s'appliquer à une même opération, dans ce cas, il sera nécessaire de les composer.

**Fonction 8** (F8) : *Application d'une interaction*
$\bigodot$ est définie de : $\mathcal{I} \times \mathcal{OP} \longrightarrow \mathcal{OP}$
Soient
- l'interaction à appliquer :
$ri = ((c_l, invocation(op_l), inputs_i, \{result\}), ca_i), inputs_i = (i^i_1, ...i^i_n)$
- l'opération sur laquelle appliquer l'interaction :
$op = (c_l, op_l, inputs, ca), inputs = (i_1, ...i_n)$ ;
$ri \bigodot op = op^+ = (c_l, op_l, inputs, \sigma_{res}(\bigoplus_{interactions}\{\sigma_i(ca_i) \cup ca\})$ où
$\sigma_i = \{(i^i_1, i_1), ....(i^i_n, i_n)\}$ et $\sigma_{res} = \{(i^{res}_1, i_1), ....(i^{res}_n, i_n)\}$ où $i^{res}_j$ désigne les variables d'entrées de l'activité composite résultat de la composition. [13]

**Fonction 9** (F9) : *Application d'un ensemble d'interactions à une opération*
$apply$ est définie de : $\mathcal{I}^n \times \mathcal{OP} \longrightarrow \mathcal{OP}$
Soit $I_l$ l'ensemble des interactions qui s'appliquent sur une même opération de nom $op_l$ associée à l'objet $c_l$ : $op = (c_l, op_l, inputs, ca))$, $inputs = (i_1, ...i_n)$
$\forall r_i \in I_l, r_i = ((c_l, invocation(op_l), inputs_i, \{result\}), ca_i), inputs_i = (i^i_1, ...i^i_n)$
$apply(I_l, op) = (c_l, op_l, inputs, \sigma_{res}(\bigoplus_{interactions}(\bigcup_{r_i \in I_l} \sigma_i(ca_i) \cup ca))$ avec
$\sigma_i = \{(i^i_1, i_1), ....(i^i_n, i_n)\}$

Par construction et grâce à la clôture et l'associativité de $\bigoplus_{interactions}$, nous avons : $apply(\{r_1...r_n\}, op) \equiv r_1 \bigodot (apply(\{r_1...r_n\}, op)) \equiv (\bigoplus_{interactions}\{r_1...r_n\}) \bigodot op$.

**Application d'une configuration** L'application d'une configuration consiste à appliquer les interactions sur les opérations désignées indirectement par les partenaires de la configuration.

**Fonction 10** (F10) : *Application d'une configuration*
$apply$ est définie de : $\mathcal{C} \longrightarrow \mathcal{OP}^n$
Soit $c = (\sigma_c, s)$, $apply(c) = \{op^+_1..op^+_p\}$ ainsi construits :
- Calcul de l'ensemble des interactions :
  $I = \bigcup_{r_i \in rules(s)}(instanciates(c, r_i))$
- Partitionnement des interactions par opérations :
  $I = I_1 \cup ... \cup I_p$, avec $I_j = \{r \in I | operation(r) = op_j\}$
  Soit $Ops = \{op_1, ..op_p\}$ les opérations à contrôler
- Application de toutes les interactions :
  $\forall op_j \in Ops, apply(I_j, op_j) = op^+_j$

12. La variable $Name$ et le terme $Name_i$ sont unifiables : soit la variable est libre (pas de valeur), soit elle a pour valeur le terme $Name_i$
13. Cette substitution pourrait être évitée en ne renommant pas les variables dans l'algorithme général, puisque dans ce cas, les substitutions des variables sont toujours inutiles.

**Exemple**

Application de la configuration : $c = (\{(a, DocAgenda), (display, TeamDisplay)\}, notification)$
Elle s'applique seulement sur l'opération $addMeeting_{Doc}$ qui devient :
$addMeeting_{Doc} = (DocAgenda, addMeeting, \{intitule\}, \{result\}, c_{addMeeting_{Doc}})$ où
$c_{addMeeting_{Doc}} = (\{ca_1\}, \{\}, \{\})$ avec $ca_1 = (\{id_1^1, id_2^2\}, \{\}, \{\})$
où $\{id_1^1, id_2^2\}$ sont simplement des copies des activités initiales.

## 6.4 Mise en œuvre : le service d'interactions

Il existe à ce jour, une mise en œuvre du service d'interactions nommée NOAH [14], qui permet de faire interagir des composants Java locaux, RMI et/ou composants EJBs.
Le service d'interactions présente les composantes suivantes :
- les interactions et les schémas sont des entités de première classe (cf. 6.4.1) ;
- des composants "interagissants" ; c'est-à-dire des objets ou composants qui ont été préparés à interagir ; leur comportement peut être modifié dynamiquement (cf. 6.4.2) ;
- un serveur d'interactions qui permet (i) de définir dynamiquement de nouveaux " schémas d'interactions " qui constituent ainsi une bibliothèque, (ii) de lier et de délier des composants par la création et la destruction d'interactions, (iii) de naviguer dans le graphe d'interactions (cf. 6.4.3) ;
- un langage de définition des schémas d'interactions (ISL : Interaction Specification Language) (cf. 6.4.4) ;
- un environnement de programmation qui sert d'interface avec le serveur d'interactions (cf. 6.4.5).

### 6.4.1 Interactions à l'exécution

Ces entités vérifient les caractéristiques suivantes :
- la cohérence des interactions est maintenue en contrôlant la réception de messages adressés aux composants,
- la définition d'un schéma d'interactions ne dépend pas des implémentations ; une interaction peut lier des objets ou composants définis sur différentes plates-formes,
- un schéma d'interactions est décrit en se basant seulement sur les services fournis par l'interface des classes de composants,
- un schéma d'interactions et une interaction peuvent être créés et détruits en cours d'exécution,
- une interaction ne peut pas contrôler des propriétés n'appartenant pas à l'interface du composant ; l'interface d'un composant lié par des interactions n'est pas modifiée même si son comportement est modifié par l'interaction,
- la composition des interactions sur un composant est basée sur la composition telle que définie en 6.2 et en 6.3.
Le serveur d'interactions stocke ainsi les schémas qui peuvent être récupérés pour modification ou analyse. Les schémas peuvent être décrits en spécialisant un schéma existant, ce type d'héritage permet d'ajouter de nouvelles règles d'interactions plus spécifiques. Il est aussi possible d'associer à un schéma une classe d'implémentation afin d'ajouter des

14. *http ://rainbow.i3s.unice.fr/noah*

attributs et des méthodes aux interactions. Notons que nous ne rencontrons pas de problèmes de fusion à ce niveau, puisque les interactions sont des objets et que les données ainsi ajoutées ne sont pas accessibles directement par les composants. Ces particularités ne sont pas développées ce mémoire car elles ne sont pas en liaison direct avec la composition sur laquelle nous avons choisi de nous focaliser.

### 6.4.2 Composant interagissant

Seuls les composants "interagissants " peuvent être liés par des interactions comportant des règles où leurs opérations apparaissent comme un pattern d'invocations. Tout composant peut cependant apparaître dans la partie action d'une règle. Un composant interagissant doit pouvoir remplir les tâches suivantes :
- composer ou (décomposer) dynamiquement des interactions,
- donner la main à un contrôle local lors de la réception de messages correspondant à une opération contrôlée. Nous nommons ces messages comme *notifiants*. Lorsqu'un composant interagissant reçoit un message qui se révèle être notifiant, l'activité composite qui lui est associée, est évaluée localement. Les appels entre les composants lors de cette évaluation sont alors directs et ne passent pas par le serveur d'interactions. Ce point est important pour mettre en place une communication directe entre composants et éviter ainsi que le serveur d'interactions devienne un goulot d'étranglement.

#### 6.4.2.1 Interopérabilité et interactions

Un des défis auquel nous avons été confrontés était de permettre à des composants situés sur des plates-formes différentes d'interagir. Les objets interagissants peuvent être des objets locaux, des objets RMI, des composants EJB,... Il s'agit de pouvoir les manipuler de la même façon tant au niveau du serveur que dans les communications directes inter-composants. Pour cela, il faut donc à la fois les désigner de façon uniforme et également minimiser les différences entre les différentes implémentations. En effet, l'analyse syntaxique, la gestion de l'arbre, la fusion sont autant d'éléments communs à toutes les versions réalisées jusqu'à présent. Pour cela, nous avons encapsulé les références dans des objets " interactingObject " qui gèrent les envois de message et présentent une même interface. Ces manipulations sont bien sûr transparentes du point de vue de l'utilisateur. Une modification du protocole de communication pour passer à RMI-IIOP devrait nous permettre de proposer l'interopérabilité avec les objets C++ que nous avions initialement dans le monde CORBA [Ber01].

#### 6.4.2.2 Mises en œuvre

Pour rendre une classe Java (locale ou RMI) interagissante, il suffit d'appliquer l'outil fourni avec le service d'interaction, "GENINT", qui modifie directement son byte-code. Le byte-code de l'objet est transformé par métaprogrammation en utilisant BCEL [Dah01] ; un mécanisme de capture de message (wrapper) est mis en place pour déléguer quand cela est nécessaire l'exécution du message à l'interprétation d'un arbre d'activités [Ber01, BMO+02].
Pour rendre une classe de composants EJB interagissant, l'utilisateur doit simplement le spécifier dans le fichier de déploiement dont la grammaire XML a été étendue. Pour Jonas [Obj08], le développeur insère la ligne suivante :
`<jonas-interaction value="true"></jonas-interaction>`.

Dans le cas d'un composant EJB, les interactions sont prises en charge par les objets d'interposition générés, au même titre que les autres services standards. Dans JOnAS, ils sont obtenus par l'outil de génération de code GenIC que nous avons modifié pour qu'il appelle l'utilitaire "GENINT" si le composant doit être interagissant. Le code généré dépend des informations décrites dans le fichier de déploiement. Le programmeur écrit donc ses composants EJB indifféremment du fait qu'ils supportent ou non les interactions. L'intégration aurait également pu se faire par modification du code de l'EJB au moyen de la métaprogrammation mais ceci au détriment de la composition du service d'interactions avec les autres services standards puisque la logique de composition de services est située au niveau des objets d'interposition.

### 6.4.3 Serveur d'interactions

#### 6.4.3.1 Pose, composition et destruction d'interactions

En cours d'exécution, le programmeur peut lier ou délier les composants en utilisant les schémas qui sont enregistrés auprès du serveur. Pour cela, il s'adresse au serveur d'interactions qui mémorise la nouvelle configuration, renvoie son nom et demande à chacun des composants liés par l'interaction et définissant des messages notifiant de fusionner les nouvelles règles instanciées qui le concernent. La prise en compte de ces modifications a lieu à compter du prochain appel.

Le retrait d'interactions consiste à demander au serveur d'interactions la destruction d'une configuration à partir de son nom. Chacun des composants déliés comportant au moins un message notifiant est prévenu afin de retirer la ou les règle(s) qui le concerne(nt) et de recalculer l'activité composite résultante. Les opérations des composants sont exécutées en mutuelles exclusions avec la pose des interactions qui ne peut donc pas 'perturber' les exécutions en cours. Dans sa thèse, Audrey Occello [Occ06] a étudié la validité du retrait ou de l'ajout d'une interaction en fonction du rôle des composants.

#### 6.4.3.2 Navigation et Analyse du graphe d'interactions

Via le serveur d'interactions, il est possible de naviguer dans le graphe d'interactions et de l'analyser. L'analyse du graphe est basée sur une représentation XML du graphe d'interactions. Le dépliage de ce graphe nous permet d'obtenir le graphe de propagation correspondant à un modèle de messages donné (destinataire et signature de l'appel donnés, paramètres non instanciés). La construction et l'analyse de ce dernier graphe permettent de détecter des circuits et des points de non-déterminisme (un même composant peut recevoir plusieurs messages dans un ordre indéterminé).

### 6.4.4 Langage de Spécifications des Interactions (ISL)

Le langage ISL présenté de façon intuitive dans les exemples précédents définit les opérateurs conditionnel, séquentiel et concurrentiel, l'attente, la levée d'exception, etc. A l'instar de l'OMG avec les approches CORBA, ce langage est indépendant des langages d'application. Il permet ainsi l'interopérabilité " interactive " des composants. Nous ne le décrivons pas davantage dans ce document et invitons le lecteur à se reporter à la

thèse de Laurent Berger [Ber01] pour plus d'informations. La figure 6.10 visualise la grammaire du langage [15].

```
Interaction     → "interaction" IDENT "(" Member { "," Member } ")"[Extensions] [ Class ] "{" Rule {"," Rule } "}"
Member          → IDENT IDENT
Extensions      → "extends" Extension { "," Extension }
Extension       → IDENT "("IDENT { "," IDENT } ")"
Class           → "implements" IDENT
Rule            → LeftSide "->" Action
LeftSide        → NotifyMsg  [ Variables ]
NotifyMsg       → Selector "." "*" | MessageDecl
MessageDecl     →
                  Selector "." "(" [ FormalParam { "," FormalParam }] ")"
FormalParam     → IDENT IDENT
Variables       → IDENT IDENT { "," IDENT IDENT }
Action          → ActionBloc ";" Action | ActionBloc "//" Action
ActionBloc      → [FrontQualif] ActionBody [BackQualif]
ActionBody      →
                  Message | Assignment | Conditionnal | "(" Action ")" | TryCatch
FrontQualif     → "[" IDENT "]" { "[" IDENT "]" }
BackQualif      → "_{" IDENT { "," IDENT }"}"
Exception       → "throw" IDENT
Assignment      → IDENT ":=" Message
Conditionnal    →
                  "if" "(" Message ")" "{ " Action "}" "else" "{" Action "}"
TryCatch        →
                  "try" "{" Action "}" "catch" (IDENT IDENT) "{" Action "}"
Message         → Invocation | CallMessage
Invocation      → Selector "." "(" [ Parameter {","Parameter} ] )
CallMessage     → Selector "." "_call"
Selector        → IDENT  "." IDENT | "this" "." IDENT
Parameter       → IDENT | Const  | "_reifiedcall"
Const           → INT | STRING | "TRUE" | "FALSE"
```

FIGURE 6.10: GRAMMAIRE ISL

### 6.4.5 Environnement de programmation

Outre les outils de génération de code présentés au paragraphe 6.4.2, plusieurs outils et interfaces graphiques ont été définis pour aider le programmeur à définir des schémas d'interactions (cf. figure 6.11, extrait de cette interface) et à contrôler son application. Ainsi la pose d'interactions peut aussi être effectuée par une interface graphique. A partir de la liste des schémas d'interactions enregistrés, l'utilisateur sélectionne le schéma d'interactions à instancier (security par exemple). Une fenêtre rappelle alors le type des composants à connecter (Object et SecurityManager) et les composants interagissant disponibles de ce type afin de poser une interaction. Cette liste présente les différents composants connus du serveur d'interaction pour avoir déjà fait l'objet d'une instanciation de schéma. Il est également possible via l'outil graphique d'étendre cette liste en définissant à l'aide d'un formulaire un autre composant qui ne serait pas déjà connu du serveur d'interaction. Dans le cas RMI par exemple ce formulaire permet de saisir l'hôte et le port d'un registre RMI ainsi que le nom du composant relativement à ce registre. L'utilisateur peut également visualiser de façon progressive le réseau d'interactions ou visualiser le graphe de propagation pour un message donné (cf. figure 6.11). Afin de faciliter la tâche du programmeur, il est également possible de visualiser la règle de réécriture associée à un composant résultat de l'ensemble des règles qui le lie.

15. Dans le cadre de ce document, nous avons travaillé sur une sous-partie de cette grammaire, en ne prenant pas en compte la gestion des exceptions, et une extension en introduisant l'opérateur *absorb*.

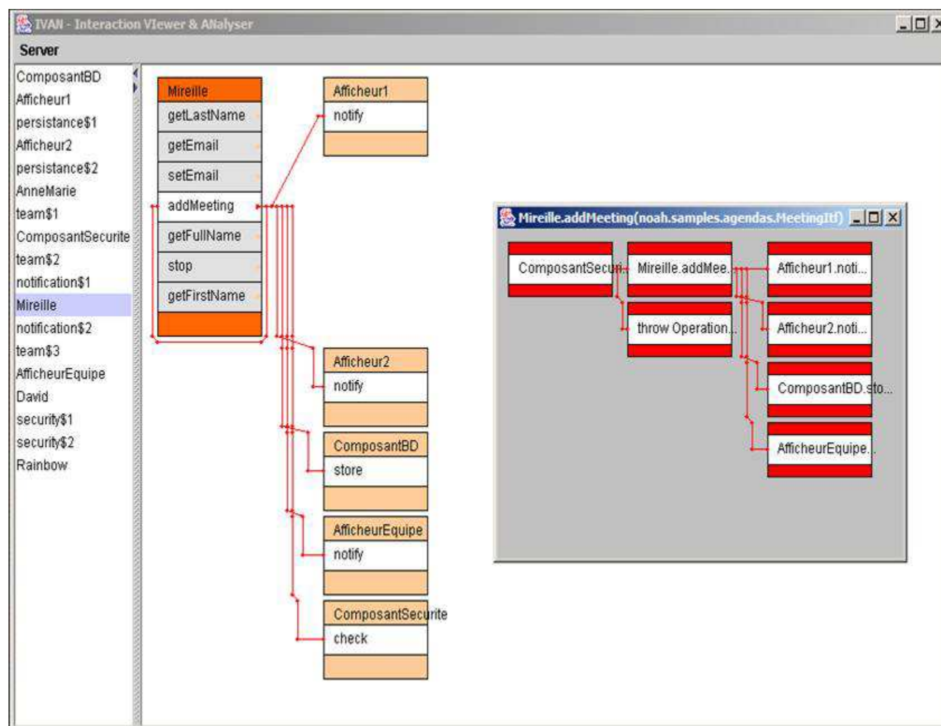



FIGURE 6.11: VISUALISATION PARTIELLE DU GRAPHE D'INTERACTIONS

## 6.5 Conclusion et perspectives

Nous avons défini un domaine d'interactions qui offre des capacités de composition répondant aux exigences que nous avions posées. Nous avons sur la base de l'algorithme présenté dans la partie I défini les fonctions nécessaires à la composition des interactions et démontré les propriétés recherchées telles que l'associativité. Nous avons montré comment nous utilisions cette composition au niveau de schémas d'interactions, puis nous avons décrit une implémentation "le service d'interactions" qui répond aux cas d'utilisation initiaux. Ce service a été déposé à l'APP et plusieurs applications ont été réalisées avec cet outil.

Les propriétés définies dans nos objectifs (cf. page 83) sont vérifiées par la composition des activités que nous avons proposée. En effet, (P1) elle ne rajoute pas de relations de précédence qui n'étaient pas explicitées dans les activités composites initiales et respecte les ordres initiaux puisque nous avons démontré la préservation des relations. (P2) Nous avons démontré l'associativité "faible" au sens où en cas d'erreur nous ne garantissons pas toujours le même résultat. (P3) La réversibilité n'est pas abordée par la formalisation mais est prise en charge au niveau de l'implémentation en recalculant l'interaction résultat en cas de retrait d'une interaction.

L'approche par métacomposition que nous avons défini nous a permis de porter un nouveau regard sur la composition des interactions en nous donnant un cadre formel.

Avant de présenter les perspectives à ce travail (cf. 6.5.3 ), nous faisons le point sur le service d'interactions et les retours d'expérience (cf. 6.5.1), puis comparons ce travail à la programmation par aspects (cf. 6.5.2).

### 6.5.1 Retours sur le service d'interactions et expérimentation

Le service d'interactions permet l'adaptation dynamique des composants par la définition de schémas, la pose et la destruction d'interactions à l'exécution. L'utilisateur exprime les interactions au niveau de l'application de manière déclarative. Dans la lignée des travaux sur la programmation par séparation des préoccupations, cette approche assure une plus grande lisibilité et évolutivité de l'application. Basée sur le langage ISL, la composition des interactions assure la commutativité et l'associativité lors de la pose des interactions. Plusieurs intervenants peuvent demander la pose ou le retrait d'interactions. Quel que soit l'ordre des requêtes le résultat est équivalent. Les incohérences entre interactions sont également détectées par ce mécanisme de composition. Les implémentations du service qui permettent de faire interagir des objets locaux et des composants RMI et EJB ont été brièvement présentées. Ces implémentations sont disponibles sur le site *http ://rainbow.essi.fr*. Elles assurent une communication directe entre les composants en interaction, évitant ainsi l'écueil des entités centralisées de gestion qui créent un goulot d'étranglement.

Nous avons eu plusieurs expérimentations sur NOAH. Elles correspondent soit à des applications métiers (Agendas collaboratifs et visualisation de systèmes experts pour aider à la gestion dynamique de bases de connaissances [DBFA+01]), soit à des outils pour la construction d'ateliers logiciels (aide active dans la manipulation de frameworks [Rap02], mise en œuvre d'intégrateurs de services [NBF04, NBF03a, NBF03b] ou contrôleurs d'interfaces utilisateurs [PDF03]). Ces différentes expérimentations nous permettent de mesurer l'utilisabilité du logiciel et son adéquation aux besoins.

En terme d'utilisabilité, l'installation de Noah ne pose pas de difficultés. Le coût d'apprentissage se situe essentiellement au niveau de la nouvelle méthodologie d'analyse des applications qui nécessite une bonne appréhension des interactions afin d'identifier le niveau de granularité adapté à l'application : quelles sont les interactions qui doivent être modélisées ? Une autre difficulté concerne le déverminage des applications réalisées pour lequel il est important d'avoir une visualisation globale du graphe d'interactions plus évoluée que la version actuelle (notion de zoom, de filtres, . . . ).
Des mesures ont pu montrer que le surcoût en temps d'exécution d'un comportement interagissant est faible du moment que les composants interagissant sont préparés ou à la compilation ou au chargement et que les composants interagissent directement, le serveur d'interactions n'intervenant qu'à la création et à la destruction des interactions et/ou schémas d'interactions. Par exemple, pour tous les composants EJBs interagissants, le surcoût lié au mécanisme d'interaction, dans le cas d'une interaction vide est celui d'un test. Dans le cas où l'interaction est non vide, le message lui-même est exécuté en utilisant la méthode d'invocation dynamique de l'API de réflexion de Java. En terme d'adéquation aux besoins, les schémas d'interactions sont adaptés à nombre de problèmes qui étaient difficiles à traiter et à maintenir tels que la composition de services et la composition d'IHM. De plus l'indépendance d'ISL par rapport aux modèles de composants sous jacents nous a permis de mettre en œuvre des applications en C++[Ber01], Java, EJBs [BMO+02] et .Net[BFCE+04, CRBFPD06].

En terme d'adéquation aux besoins, dans les applications dédiées à des domaines spécifiques, systèmes experts par exemple, nous avons fourni des bibliothèques de schémas qui facilitait l'adaptation des applications par les utilisateurs finaux des systèmes ex-

perts. Nous avons ainsi constaté que l'expression de schémas génériques a un apport indéniable car le même schéma se retrouve utilisé avec des types différents. L'exploitation des schémas pour fournir des outils d'intégration tels que des intégrateurs de service ont été proposés dans la thèse de Olivier Nano [Nan04] et l'usage a été renforcé avec les travaux autour des aspects d'assemblage autour de WCOMP [CFWBFT+05]. Les travaux en cours de Clémentine Nemo autour des patterns [NBFR08] renforcent également cette approche des schémas d'interactions (même sous une forme différente) comme un outil d'adaptation pour autant que la sémantique de la composition soit claire.
Cependant nous avons pu identifier certaines insuffisances dans la puissance d'expression d'ISL et la nécessité d'avoir des interactions génériques réutilisables selon les contextes. Ces différents points sont développés dans les perspectives.

### 6.5.2 Interactions et programmation par aspects

La programmation par aspects supporte en partie les interactions. Cependant nous pouvons noter plusieurs différences. Pour gérer les "interactions" entre aspects, ces systèmes proposent à l'utilisateur de spécifier la composition des aspects en utilisant par exemple une relation d'ordre entre les aspects, alors que nous défendons l'idée qu'une réelle séparation des préoccupations impose une composition "automatique", indépendante de l'ordre de déclaration des interactions, et respectant l'ordre partiel latent aux interactions. La plupart des travaux ont jusqu'à présent porté sur l'intégration des aspects au niveau des classes, ce qui nous différencie également. Enfin ces travaux définissent des langages qui adressent des environnement dédiés, et n'abordent pas l'interopérabilité ou la distribution des interactions.
Le modèle des interactions se différencie donc de AspectJ et d'autres travaux tels que [PSDF01, PSDC06], par une approche indépendante langage, la gestion de composants hétérogènes et distribués, un positionnement au niveau des instances et une composition qui ne dépend pas de l'ordre de déclaration.
Nous donnons ci-après une comparaison plus précise avec AspectJ.

#### Comparaison avec AspectJ

Le langage AspectJ [KHH+01] est probablement le plus connu des langages d'aspects. Il présente la caractéristique d'offrir au programmeur de l'aspect de nombreux types de points de " jonctions ", tels que la réception de message (comme nous) mais aussi l'envoi de message (éventuellement dans une méthode donnée) ou le contrôle au niveau des instances. Le code de l'aspect est définissable dans des "advice" qui se placent avant, après ou autour du point de jonction. Cette partie de l'aspect correspond dans notre approche à la partie droite de la règle. La composition des aspects est basée sur une relation d'ordre entre les opérateurs *before* , *after* et *around* et l'utilisation de règles de précédences entre les aspects. Les figures 6.12 et 6.13 visualisent au travers d'exemples ces différences. En particulier, alors que l'approche ISL n'introduit pas d'ordre entre les parties "before" exprimées séparément, AspectJ utilise l'ordre de déclaration des aspects pour ordonnancer les activités entre aspects. Cette ordre a d'autant plus d'importance que la présence ou non de l'instruction *proceed* dans une partie *around* peut interdire l'exécution d'un autre aspect, ceci en fonction de l'ordre de priorité entre les aspects. Cette situation peut correspondre en ISL à la présence d'un *absorb* ou d'un *delegate*, cependant le comportement est différent comme le visualise la figure 6.13. Alors que dans AspectJ, un aspect peut absorber tous les autres aspects, s'il est prioritaire, dans notre approche, le comportement

est différent puisque l'absorption est explicite, et n'intervient que sur les successeurs. Inversement, l'absence d'ordre entre les interactions ne permet pas d'adapter le résultat de la composition. En conclusion actuellement les comportements de composition de AspectJ et de ISL sont différents et ne peuvent pas être simulés les uns par les autres.

### 6.5.3 Perspectives

Les perspectives qui suivent seront développées dans d'autres cadres que celui d'ISL. En effet, si lors de l'invention du langage, la démarche était originale, aujourd'hui de nombreux travaux traitent de ce problème, le plus souvent sous la forme d'aspects au niveau des classes. Nous ne pensons pas qu'il s'agisse de la même solution. Notre démarche recherche une composition indépendante langage, intègre l'interopérabilité et surtout repose sur des mécanismes de composition associatifs et commutatifs. Néanmoins, nous nous intéressons aujourd'hui à d'autres aspects de la composition en particulier dans le cadre des workflows et des transformations de modèles. Nous avons donc fait le choix de stopper le développement de Noah et le langage a été repris dans plusieurs domaines (IHM, aspects d'assemblages) pour être adapté sous d'autres formes. Pour ma part, j'ai fait le choix de me concentrer sur les modèles de composition (la partie modélisation présentée précédemment est originale en ce sens) et d'appliquer ces modèles à d'autres domaines que les composants.

Dans ce contexte, parmi les extensions des travaux sur les interactions, nous nous intéressons à l'introduction d'autres activités pivots qui offriraient donc des comportements différents lors de la composition, avec comme objectif de préserver les propriétés de composition et de ne pas introduire de redondance. En particulier, la nécessité de pouvoir retourner une valeur à l'appelant alors que la résolution d'une interaction n'est pas terminée, nous amène à introduire une activité "reply". Celle-ci sera étudiée dans un contexte un peu différent dans l'application suivante sur les compositions d'orchestration (cf. §7), mais son exploitation dans un contexte incluant les conditionnelles est une de nos perspectives. De même la gestion des exceptions est une extension que nous envisageons par une extension du métamodèle MM4CA.

La formalisation de la composition des aspects selon notre démarche de métamodélisation a été amorcée en associant aux activités les informations "before" ou "after" et en gérant des priorités.
Nous n'avons pas abordé l'appariement entre les points de coupe et leur composition. Or, c'est un des points difficiles de la programmation par aspects qui, de notre point de vue, induit une partition de l'espace en fonction des recouvrements entre les aspects, comme nous l'avons abordé dans la thèse d'Olivier Nano. De récents travaux étendent la définition des points de coupe en introduisant les notions de signatures fournies et requises lors de la définition des *advices around*[FSJ08]. Nous intégrerions ces points dans notre algorithme au niveau des informations portées par les activités et par la construction d'activités explicitant le partitionnement.
Dans sa thèse Pessemier montre combien le séquencement d'aspects est problématique [Pes07, chapitre 8] ce qui rejoint le besoin de commutativité et d'associativité que nous défendons. Il propose une approche basée sur la structuration des aspects sous la forme de composants et une approche programmative de l'introduction des aspects. De même

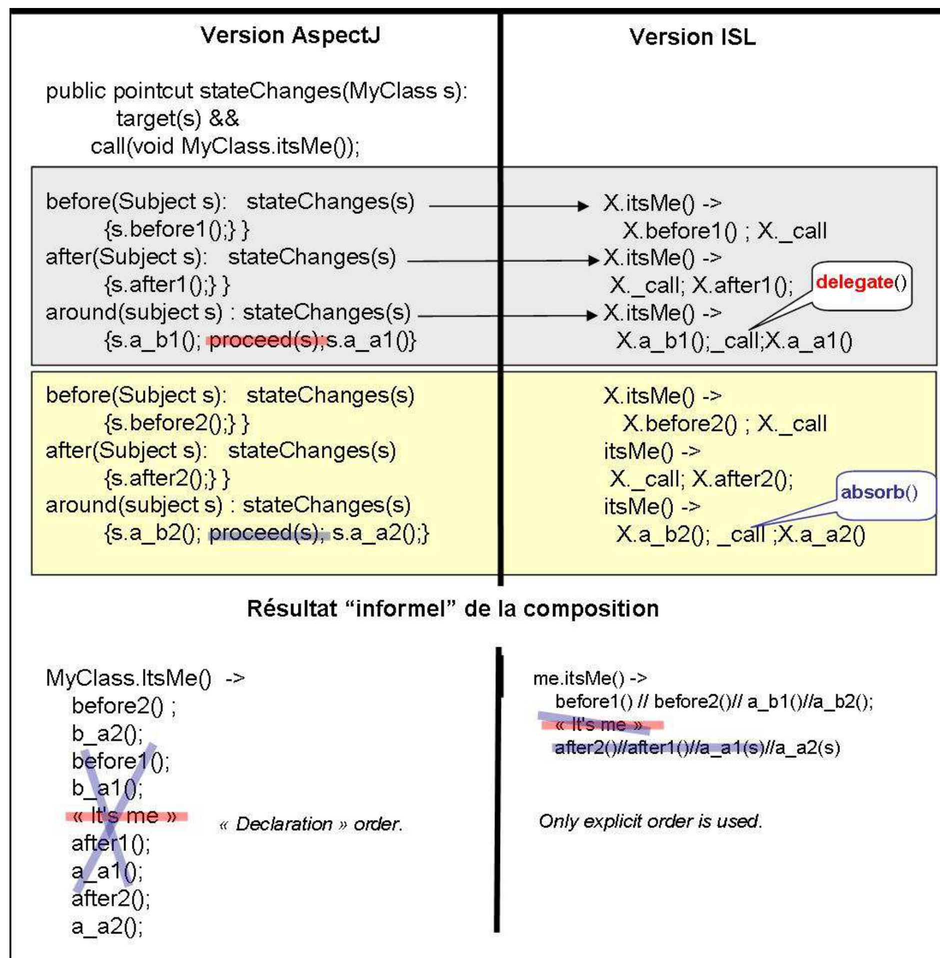
| Version AspectJ | Version ISL |
|---|---|
| public pointcut stateChanges(MyClass s):<br>target(s) &&<br>call(void MyClass.itsMe()); | |
| before(Subject s): stateChanges(s)<br>{s.before1();} } | X.itsMe() -><br>X.before1() ; X._call |
| after(Subject s): stateChanges(s)<br>{s.after1();} } | X.itsMe() -><br>X._call; X.after1(); |
| around(subject s) : stateChanges(s)<br>{s.a_b1(); proceed(s);s.a_a1()} | X.itsMe() -><br>X.a_b1(); _call ;X.a_a1() |
| before(Subject s): stateChanges(s)<br>{s.before2();} } | X.itsMe() -><br>X.before2() ; X._call |
| after(Subject s): stateChanges(s)<br>{s.after2();} } | X.itsMe() -><br>X._call; X.after2(); |
| around(subject s) : stateChanges(s)<br>{s.a_b2(); proceed(s); s.a_a2();} | X.itsMe() -><br>X.a_b2(); _call ;X.a_a2() |

**Résultat "informel" de la composition**

| | |
|---|---|
| MyClass.ItsMe() -><br>before2() ;<br>a_b2();<br>before1();<br>a_b1();<br>« It's me »<br>after1();<br>a_a1();<br>after2();<br>a_a2();<br><br>*« Declaration » order.* | me.itsMe() -><br>before1() // before2()// a_b1()//a_b2();<br>« It's me »<br>after2()//after1()//a_a1(s)//a_a2(s)<br><br>*Only explicit order is used.* |

*Nous représentons le code initial de la méthode itsMe par "It's me".*
*La composition en AspectJ utilise l'ordre des déclarations (à gauche, le deuxième aspect en jaune est considéré comme prioritaire) tandis que l'approche ISL (à droite) n'introduit pas d'ordre entre les aspects. En particulier, la décomposition proposée pour ISL ici ne tient pas compte de la précédence d'un before sur un around. Pour cela nous aurions dû écrire une seule règle pour définir l'aspect du haut, par exemple :*
`X.before1();X.a_b1();_call;X.a_a1();X.after1()`

FIGURE 6.12: COMPARAISON DES COMPOSITIONS PAR L'EXEMPLE ENTRE ASPECTJ ET ISL (*proceed*)




*L'absence de proceed dans une partie around d'un advice en AspectJ interdit l'exécution des autres around.*
*- Dans cet exemple, nous visualisons en rouge l'absence de proceed dans le* $1^{er}$ *aspect en haut, qui est interprété à droite comme l'utilisation d'un delegate. L'absence du proceed dans l'aspect de priorité moindre conduit à la seule disparition de l'appel à la méthode initial. Le comportement du delegate joue le même rôle quelque soit l'ordre des aspects.*
*- L'absence du proceed dans le deuxième aspect, donc celui de plus forte priorité, a pour conséquence d'"absorber" le premier aspect. Seul les activités du deuxième aspect sont exécutées. Ce comportement est pris en charge par l'opérateur absorb dans ISL.*
*Tandis qu'en AspectJ, l'ordre des aspects permet d'exprimer différentes compositions, en ISL, nous avons fait le choix d'une définition d'opérateurs autorisant une composition indépendante des ordres de composition.*

FIGURE 6.13: COMPARAISON DES COMPOSITIONS PAR L'EXEMPLE ENTRE ASPECTJ ET ISL (*delegate* ET ORDRE DE COMPOSITION)

dans [PDS05], les auteurs proposent d'expliciter des contraintes sur les aspects et l'ordre entre les aspects et de vérifier que cet ordre respecte bien les contraintes. Est-il possible de concilier une approche déclarative et automatique de composition telle que nous la proposons et impérative comme proposée par FAC-AOKell ou CompAr ? Ce point fait l'objet de nos perspectives en étudiant le même exemple mais dans un monde workflow.

Les travaux sur les connecteurs [Car03] mettent en avant la nécessité d'une séparation entre la mise en œuvre des composants et les communications du composant avec son environnement. Jusqu'à présent, les interactions sont basées sur un protocole par envoi de message. Une extension du modèle pour intégrer une communication par événement, sans perdre les propriétés de composition, est à l'étude dans l'équipe autour de Wcomp [CFWBFT+05].
A l'instar des travaux sur les ADLs, il nous semble important de vérifier la cohérence globale du graphe de propagation. Pour l'instant nous ne vérifions que la cohérence locale du graphe des interactions (toutes les interactions locales à un composant sont " compatibles "). L'étude sur la cohérence du graphe d'interactions a été amorcée en supposant le graphe d'interaction stabilisé à un moment donné. Les résultats issus des ADLs devraient être applicables. Nous nous intéressons à ces problèmes actuellement dans les workflows par transformation vers des algèbres de processus [Pou07] ; ce travail est une de nos perspectives à court terme.

# CHAPITRE 7

# Composition d'orchestrations

## Contexte de recherche et contributeurs

*Ce travail a été commencé par le DEA de Clémentine Nemo[Nem06, NBFKR07]. Une approche différente est aujourd'hui suivie par Sébastien Mosser[MBFR08]. Son travail est présenté en perspective de ce chapitre. Une application qui a guidé la réalisation de ce travail vient d'une collaboration avec Johan Montagnat et Tristan Glatard[NGBFM07]. Enfin notons que les origines de ce travail remontent à des remarques de Yves Caseau sur le rapprochement des interactions avec des workflows lors de la revue du projet Rainbow en 2002 et des réflexions de David Emsellem sur l'application des interactions aux orchestrations de web services. La problématique et les esquisses de solutions proposées ici sont issues de nos collaborations dans le cadre de différents projets industriels (Thèse avec* DCNS *Toulon, Projet* RNTL FAROS *...). Une plate-forme est en cours de développement pour étayer ces différents points [JMBF07, JMBFN07]. J'ai volontairement circonscrit ce domaine au travail mené dans le cadre du DEA de Clémentine Nemo pour ne pas interférer avec la thèse en cours. Ce domaine est donc plus prospectif que le précédent.*

## Plan de ce chapitre

Dans un premier temps (cf. §7.1), nous précisons nos objectifs relativement à la composition dans le cadre d'orchestrations ayant des invocations en commun.

La composition d'orchestrations constitue un domaine de composition que nous formalisons relativement à MM4CA(cf. §7.2).

Nous formalisons alors la composition dans le domaine des orchestrations et développons les propriétés respectées par cette composition (cf. §7.2.8). Nous présentons ensuite la mise en œuvre de cette modélisation (cf. §7.3).

Enfin nous concluons ce chapitre par un retour sur expérimentation et le positionnement de notre travail vis-à-vis d'autres travaux, ce qui nous amène à expliciter nos perspectives relativement à ce travail.

## 7.1 Domaine applicatif et objectifs

Le concept de service est le sujet de définitions très variées. Nous reprenons ici la définition donnée par le consortium Oasis dans [MLM+06] [1] : *A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description... A service is opaque in that its implementation is typically hidden from the service consumer except for (1) the information and behavior models exposed through the service interface and (2) the information required by service consumers to determine whether a given service is appropriate for their needs...*
Les interfaces et les données exhibées par le service sont exprimées en termes métiers. Les aspects technologiques ne sont plus essentiels car les services sont autonomes, c'est-à-dire qu'ils sont indépendants du contexte d'utilisation ainsi que de l'état des autres services, et interopèrent via des protocoles standardisés. Un service définit une entité logicielle (*e.g.,* ressource, application, module, composant logiciel, etc.) qui communique via un échange de messages et qui expose un contrat d'utilisation. De façon similaire aux approches par objet ou par composant, l'approche service cherche à fournir un niveau d'abstraction encore supérieur, en encapsulant des fonctionnalités et en permettant la réutilisation de services déjà existants. La propriété de couplage faible que cette approche induit est à l'origine des architectures agiles que sont les architectures orientées services.
Les architectures orientés Services (SOA, [MLM+06]) facilitent l'exposition, l'interconnexion, la gestion et l'évolution d'applications à base de services. Les orchestrations sont au cœur de ces architectures en supportant la construction d'applications à partir de fonctionnalités de base. Créer des compositions de services signifie ordonner les invocations aux opérations, router les messages, modifier les paramètres et gérer les exceptions. Plusieurs langages de composition de composition de services sont définis (WSCI [SISM02], WSBPEL [JEA+07], SCUFL [OAF+04]).
Dans le cadre des Services Web, nous nous intéressons plus particulièrement aux orchestrations. Les orchestrations sont définies par le consortium W3C (*W3C Glossary*) comme "le modèle des interactions que doit respecter un agent Service Web pour atteindre son but". Cependant, même si les orchestrations sont un support à une programmation incrémentielle par la construction de nouveaux services par assemblages [CP05, BJPK+05], elles n'offrent pas de support à leur propre composition. Ce point est d'autant plus critique que les applications à base de services évoluent très rapidement et que le développement collaboratif, en particulier par séparation des préoccupations est indispensable à l'élaboration des gros projets.

### 7.1.1 Enjeux de la composition d'orchestrations

De nombreux travaux dans l'industrie et la recherche portent sur la composition des services. On peut distinguer deux approches. L'une vise à proposer des outils pour modéliser les compositions de services[BCG07, Rub08, PS05], l'autre s'intéresse à une composition automatique des services en fonctions des annotations sémantiques qui leur sont associées [MHS06, RDHS06].
Ainsi, associée à l'idée d'un réseau de services vient l'idée que l'évolution d'une application ne nécessite plus d'acquérir la globalité de son fonctionnement [GEM06]. La compo-

1. la définition est donnée en anglais pour éviter toute *interprétation* lors de la traduction.

sition des services n'aurait donc pas à prendre en compte le fait qu'un service correspond ou non à une orchestration. En conséquence, à notre connaissance peu de travaux s'intéressent à la composition d'orchestrations.
Or la réalité n'est pas si simple et la composition d'orchestration de services est une tâche critique, qui intervient lors d'intégrations d'applications ou dans la construction d'applications de grandes tailles. Cette tâche, faîte manuellement, est cause d'erreurs et chronophage. En effet elle suppose de répondre aux mêmes exigences de validité et d'optimisation des orchestrations que les services correspondent ou non à des orchestrations.

**Validité des orchestrations** La définition des orchestrations est une manière d'expliciter la construction d'applications alors qu'une partie de celle-ci est en fonctionnement. Dans ce contexte, différents travaux s'intéressent à valider les assemblages ainsi obtenus en proposant des conversions vers des algèbres de processus [CCCV05, Mar05] pour vérifier des propriétés telles que la "compatibilité des services" et le remplacement des services en prenant en compte à la fois les aspects "syntaxiques" et le comportement des services. Mais ce travail exige une formalisation de l'ensemble des services mis en jeu dans une orchestration. Ainsi la sûreté des assemblages repose sur une expression plus complète que l'ensemble des fonctionnalités offertes par un service et en particulier l'expression du protocole relatif au service. Ce point est particulièrement critique lorsque les services sont des orchestrations.

**Optimisations** En intensifiant le nombre des services mis en jeu dans les applications à base de services, on constate des besoins d'optimisation au niveau des orchestrations par exemple par le regroupement des services en interactions dans des orchestrations dédiées, éliminant ainsi des références vers l'orchestration globale de l'application [CCMN04]. De même l'adaptation fonctionnelle séparée de certains services d'une application distribuée peut conduire à des flots d'appels non optimisés : répétitions d'appels vers un même service ou des services équivalents (mise à jour d'un affichage, service de recherches appelés plusieurs fois dans une propagation), introductions de médiateurs inutiles ou augmentation des appels vers un service non prévu à cet effet [Nem06]. La composition des applications à base de services suppose donc la prise en charge d'optimisations des assemblages résultants.
Ainsi un problème récurrent lors de la composition des web services est que le contenu exact d'un service correspondant à une orchestration n'est pas "connu" au sens où son fournisseur ne veut pas révéler son contenu. Cependant, pour assurer la qualité de service ou même la détection d'inter-blocage, cette connaissance est nécessaire. En situant le travail sur la composition des orchestrations au sein d'une entreprise nous évitons cet écueil. Mais à la manière de Pankratius et autres [PS05], il est également possible de limiter la "vue donnée" sur l'orchestration à un ensemble d'activités mettant en jeu en particulier des services communs. Ainsi de la même manière que se développent des lignes de produits, des workflows sont définis comme des services réutilisables et standardisés. Or, la composition des workflow eux-même engendre différentes difficultés lorsque ceux-ci font en particulier référence à des services communs[NGBFM07] :

- Problèmes métiers - Par exemple, la composition d'orchestrations portant sur la réservation d'un hôtel et d'un vol pourra engendrer l'envoi de plusieurs factures alors que l'envoi d'une seule facture aurait été préférable ;
- Problèmes économiques - Certains services sont payants ; il est alors plus intéressant d'adresser de tels services, comme par exemple un service bancaire, en soumettant

l'ensemble des données une seule fois plutôt que de payer plusieurs fois pour la même information ;
- Problèmes de performance - Par exemple, dans le cadre d'applications scientifiques, il peut être préjudiciable d'adresser plusieurs fois un même service dont le temps d'exécution est long ; si les données sont différentes, il peut être intéressant de soumettre l'ensemble des données en une seule fois.

En résumé, la composition par assemblage (approche boites noires) répond mal au partage de contexte entre des sous-processus, à l'optimisation des workflows, à la gestion des exceptions, à la répartition des contrôles d'accès, au contrôle de la qualité de services, aux calculs de coûts, etc (cf. §1.2). La composition des orchestrations implique de gérer les incompatibilités entre les modèles de données, les contraintes de séquencement dans l'ordre des invocations, ...
Nous nous sommes intéressés à cette problématique et nous la présentons à présent sur un exemple.

### 7.1.2 Exemple de composition d'orchestrations

Nous prenons ici un exemple basé sur l'application dites du "Bronze Standard" qui vise à évaluer statistiquement l'exactitude d'algorithmes de recalages d'images médicales en l'absence d'une réalité terrain (par exemple le cerveau du patient)[NGBFM07]. Le recalage consiste à déterminer une transformation géométrique permettant de passer d'une image (dite source) à une autre image (dite cible) acquises indépendamment, et à construire l'image de sortie. Les données d'entrées d'un tel algorithme sont donc une paire d'images et une liste de paramètres spécifiques à l'algorithme, et la donnée de sortie est une transformation. Le Bronze Standard utilise de nombreux couples d'images et de nombreux algorithmes de recalage pour déterminer la précision des algorithmes en calculant la distance de son résultat au bronze standard établi en calculant la moyenne des transformations obtenues. Le nombre d'algorithmes n'est pas fixe ; plus d'algorithmes sont fournis plus la procédure est précise. Aussi l'orchestration du Bronze Standard nécessite d'être recomposée en fonction des algorithmes utilisés.
La figure 7.1 visualise deux orchestrations basiques (gauche et centre) qui ont été définies par les développeurs. Chacune est constituée d'une partie algorithme (partie haute), connectée à une partie évaluation (partie basse) en charge d'évaluer la précision de l'algorithme. La partie algorithmique inclut la partie recalage (**C**rest**M**atch à gauche et PF**M**atch au milieu). Les deux commencent par un appel au service commun CL qui à partir d'un couple d'images et un paramètre numérique retourne deux fichiers (*CrestLines*) qui sont les entrées des services CM ou PFM. Ensuite les orchestrations diffèrent : l'algorithme CM nécessite en plus des deux fichiers *CrestLines* les deux images à traiter tandis que l'algorithme PFM prend en plus une matrice correspondant à une transformation initiale et une constante. Finalement les deux algorithmes produisent une matrice de transformation et un commentaire qui sont transférés à une partie commune d'évaluation. La partie évaluation est composée de 3 services liés séquentiellement. Le premier service *Convert* convertit la matrice de transformation résultante de l'algorithme en un vecteur. Le format de conversion dépend de la sortie de l'algorithme de recalage. Le vecteur est alors concaténé aux noms de images en entrées et au commentaire, puis écrit par le service *Write* dans un fichier résultat. Lorsque toutes les données ont été traitées par le service *Write*, le service *Eval* détermine la précision de l'algorithme, qui est le résultat de l'orchestration.

Pour composer les deux orchestrations et obtenir l'orchestration présentée à droite de la figure 7.1, le développeur doit déterminer les données communes, les invocations de services surchargées, les valeurs de retour. Si nous considérons plus d'orchestrations et de taille plus importante, la tâche est particulièrement difficile. Si de plus les orchestrations initiales sont modifiées, la tâche doit à nouveau être répétée Les besoins d'une composition automatique des orchestrations sont alors évidents. Cependant certaines décisions dépendent de la sémantique des services et ne peuvent être prises que par le développeur des orchestrations comme par exemple l'utilisation de la sortie de CM comme un paramètre d'initialisation de PFM, invoquer les deux *Eval* et *Write* séparément, ....

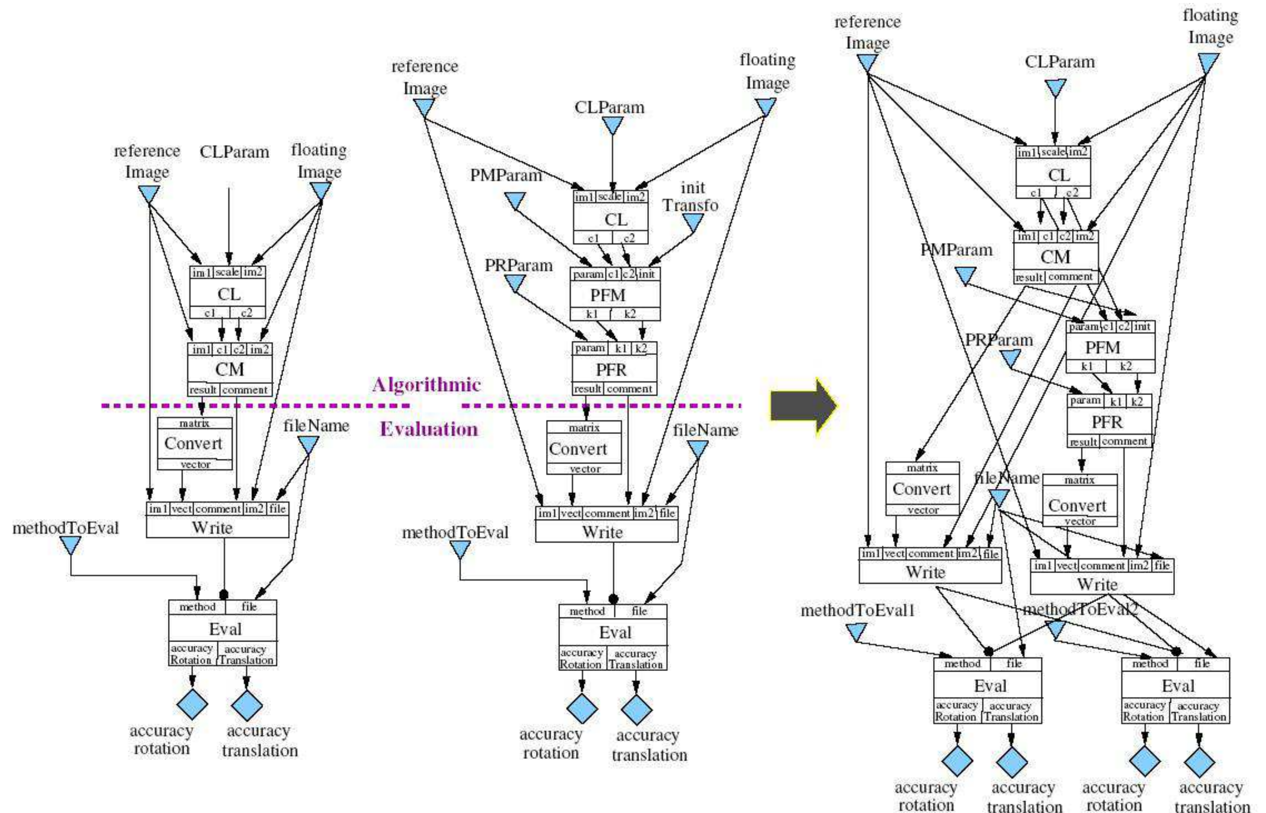



FIGURE 7.1: ORCHESTRATIONS CM (GAUCHE), PFM (MILIEU) ET COMPOSÉE (DROITE).

Nous proposons donc de définir la composition des orchestrations comme un processus semi-automatique basé sur (i) la détection des points de fusion potentiels (invocations surchargées, variables d'entrées de l'orchestration, instruction de retour) (cf. figure 7.2) (ii) la proposition d'opérations de fusion en fonction de l'orchestration, (iii) l'unification automatique des variables dont l'identité conceptuelle peut être dérivée d'étapes précédentes de fusion.

### 7.1.3 Notre approche : composition par reconnaissance des activités surchargées

Nous abordons dans cette partie la composition d'orchestrations comme une application de l'algorithme général présenté dans la partie I "modélisation des compositions d'acti-

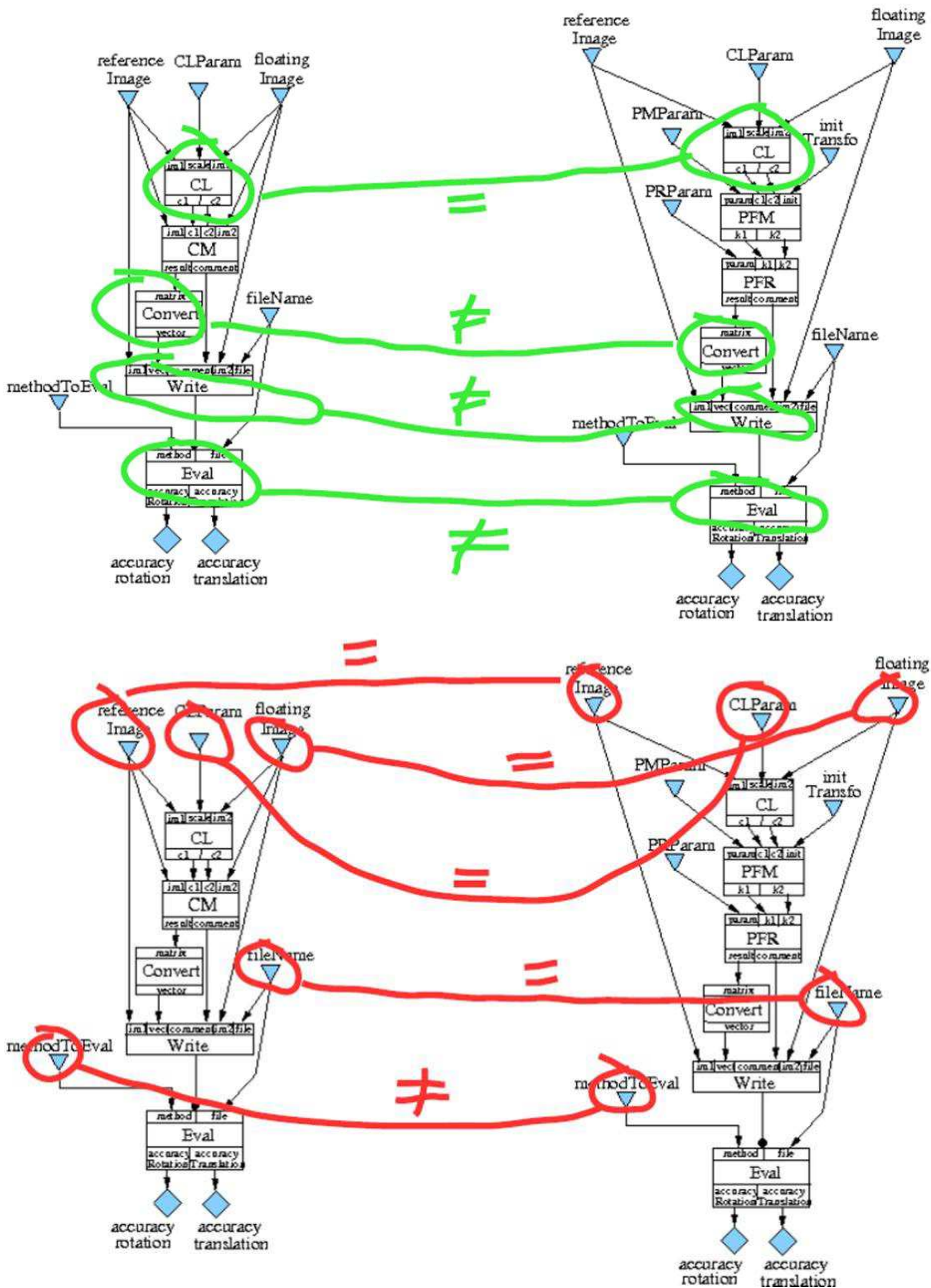



FIGURE 7.2: DÉTECTION DES PAIRES DE FUSION ET DÉTERMINATION DES OPÉRATIONS DE FUSION

vités". Outre l'intérêt applicatif de ce travail il présente une déclinaison interactive de l'algorithme. La composition des orchestrations ne peut pas en effet être complètement automatisée. Nous avons donc identifié différents points de choix et relativement à la vérification des propriétés qui nous semblent essentielles, nous guidons l'utilisateur dans un processus complexe de composition.

Une orchestration correspond "naturellement" à une activité composite. Lors de la composition d'activités composites correspondant à des orchestrations, nous voulons *(i)* éviter les accès concurrents en lecture et écriture à une variable, *(ii)* optimiser l'orchestration résultante (par exemple en reconnaissant les appels multiples à un même service ou en partageant des données), *(iii)* respecter les ordres d'invocations entre les services, *(iv)* assurer l'unicité de l'instruction de retour. L'*associativité* est également intéressante. Elle permet d'envisager la composition des orchestrations de manière collaborative indépendamment des temps de compositions. Cependant, pour l'atteindre, il nous faudrait limiter davantage les possibilités de l'utilisateur. Nous avons choisi pour l'instant la flexibilité.

L'interprétation dans le domaine des orchestrations détermine l'ensemble des couples de fusion potentiels sur la base d'une simple comparaison (syntaxique) des activités. Les possibilités de composition sont alors nombreuses, et il n'est pas possible sur la base du seul modèle des activités de toujours automatiquement décider de la composition. A la manière de M. Didonet dans [FV07], nous proposons donc à l'utilisateur de choisir parmi les compositions possibles. Lorsque le choix est difficile, il peut être retardé. La résolution d'un autre ensemble de fusion pouvant réduire les choix. Par exemple, la composition des entrées de deux orchestrations (activité receive) peut être retardée en fonction des composition des sous activités, qui unifieront alors des variables. La composition elle-même est automatisée en fonction des règles de composition choisies et de l'algorithme général.

## 7.2 Formalisation et composition dans le domaine des orchestrations

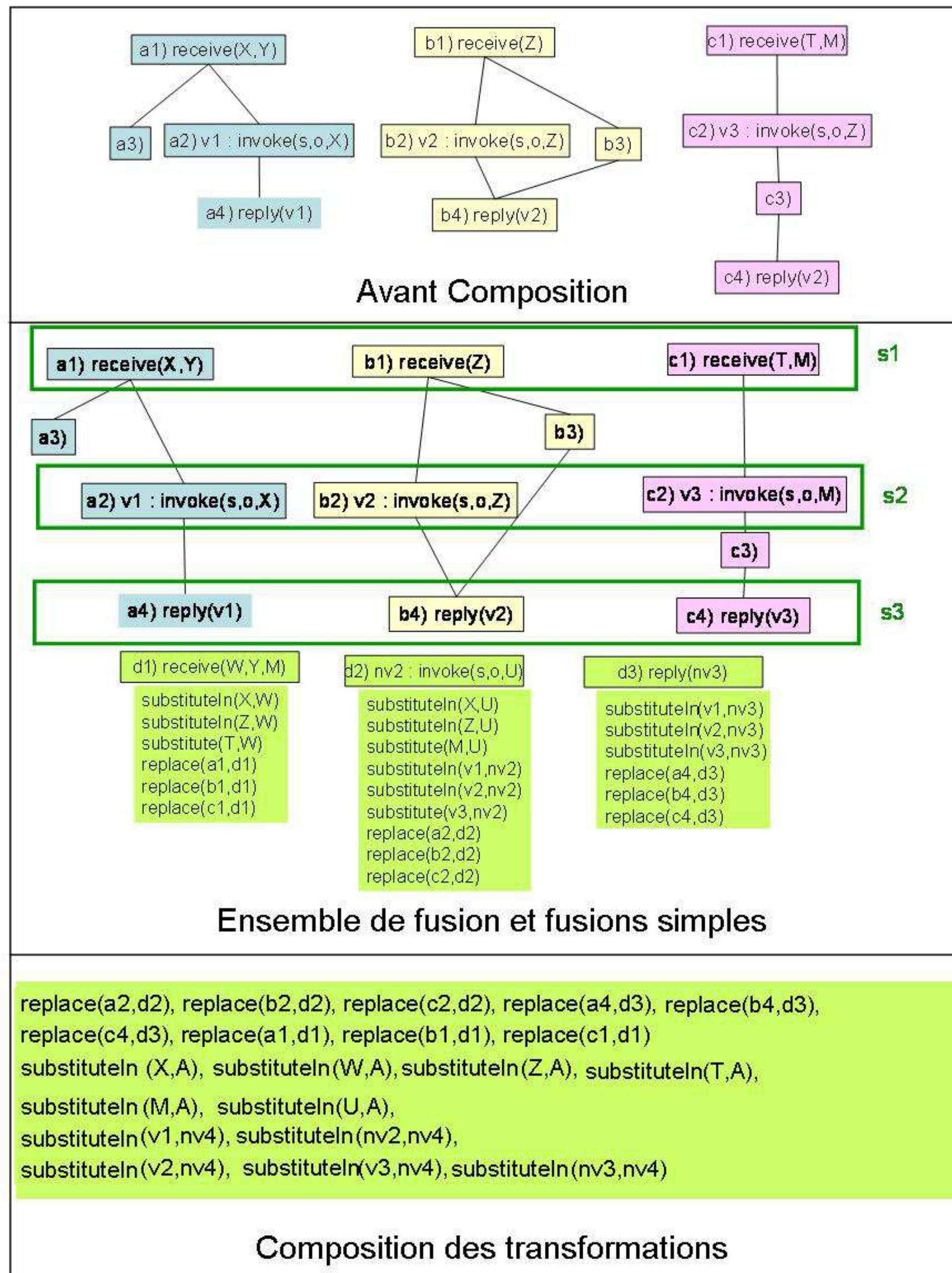
Contrairement aux exemples précédents, cette fois-ci tous les ensembles de fusion ne seront pas résolus à chaque étape. En conséquence, il y a itération : des fusions simples sont déterminées, les transformations sont appliquées et les ensembles de fusion sont recalculés.

Les figures 7.3 et 7.4 visualisent un exemple de composition d'orchestrations pas à pas. Nous posons les bases de cette composition dans ce qui suit.

### 7.2.1 Domaine applicatif et contraintes sur domaine

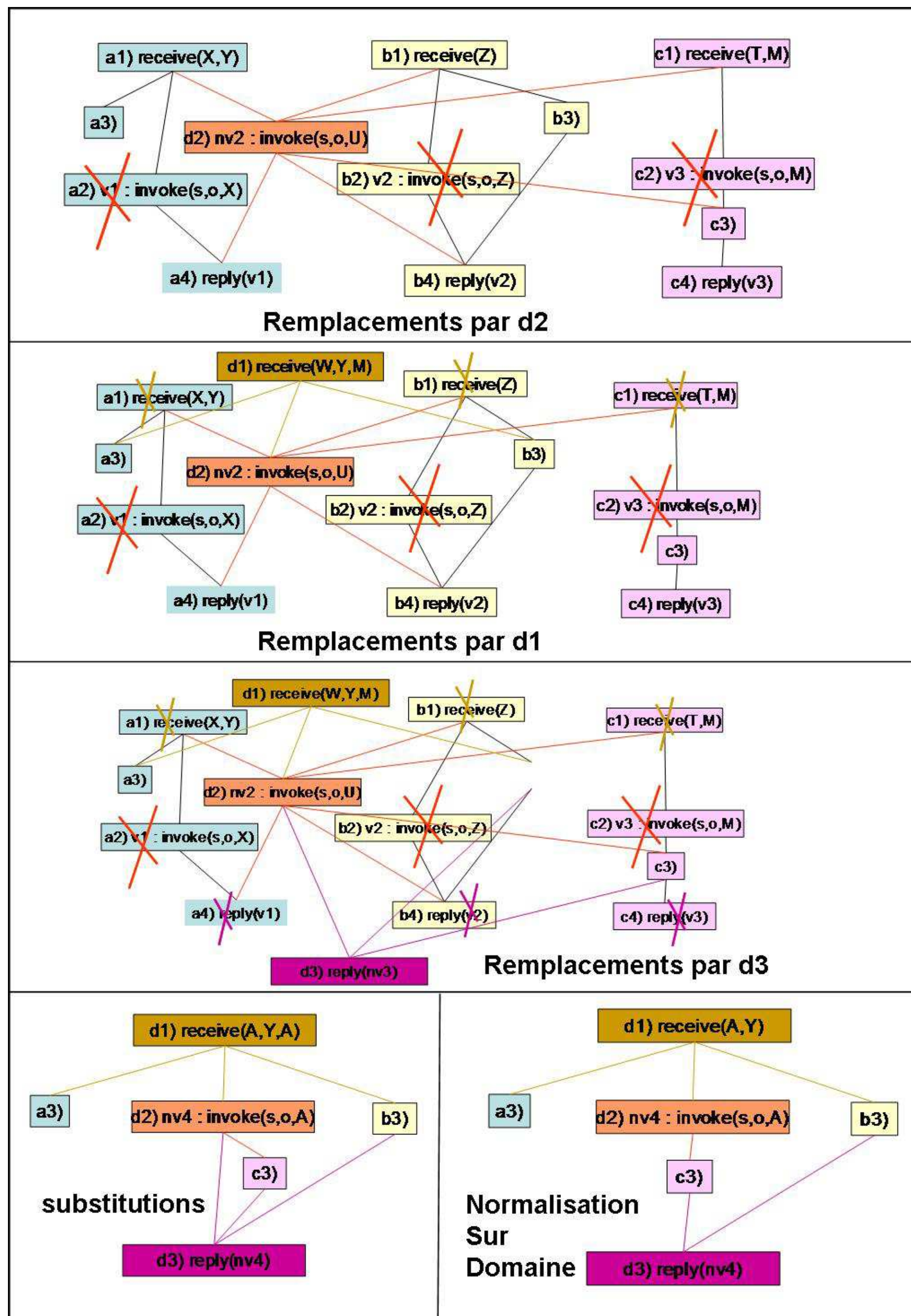
Une orchestration est une activité composite dont les activités simples sont construites selon la définition du domaine des orchestrations tel que défini ci-après. Nous n'utilisons pas la relation conditionnelle dans ce domaine.

La figure 7.5 présente l'extension du métamodèle MM4CA.




*Les activités sont identifiées par un identifiant, comme dans les exemples de la partie I. Mais nous visualisons également les informations et variables des activités pivots pour expliciter les mécanismes de composition.*
*Dans cet exemple les activités* receive *(ensemble de fusion noté s1) sont fusionnées par l'unification des variables X, Z et T, tandis que Y et M restent indépendantes ; cette fusion crée l'activité d1 et les transformations associées visualisée sous d1 ;*
*les invocations simples (ensemble de fusion noté s2) sont fusionnées en une invocation simple d2 avec l'unification des variables X,Z,M ce qui engendre l'ensemble des transformations visualisées sous d2*
*et les activités* reply *(ensemble de fusion noté s3) sont également unifiées avec l'unification des variables v1, v2, v3.*
*La composition des transformations donne l'ensemble des transformations visualisé en partie basse de la figure.*
*La figure suivante visualise l'application des transformations.*

FIGURE 7.3: EXEMPLE DE COMPOSITIONS D'ORCHESTRATIONS : DE LA DÉTECTION DES ENSEMBLES À LA COMPOSITION DES TRANSFORMATIONS.




*Les transformations* `replace` *sont tout d'abord appliquées. Elles reproduisent sur l'activité de remplacement, les relations d'ordres existants sur l'activité remplacée. Nous visualisons successivement le remplacement par d2 de a2, b2 et c2, puis celui par d1 de a1,b1 et c1 et enfin celui par d3 de a4, b4 et c4.*
*Les substitutions des variables sont alors appliquées sur les seules activités restantes. Finalement la normalisation sur domaine élimine les relation d'ordre redondantes.*

FIGURE 7.4: EXEMPLE DE COMPOSITIONS D'ORCHESTRATIONS : APPLICATIONS DES TRANSFORMATIONS ET NORMALISATION SUR DOMAINE

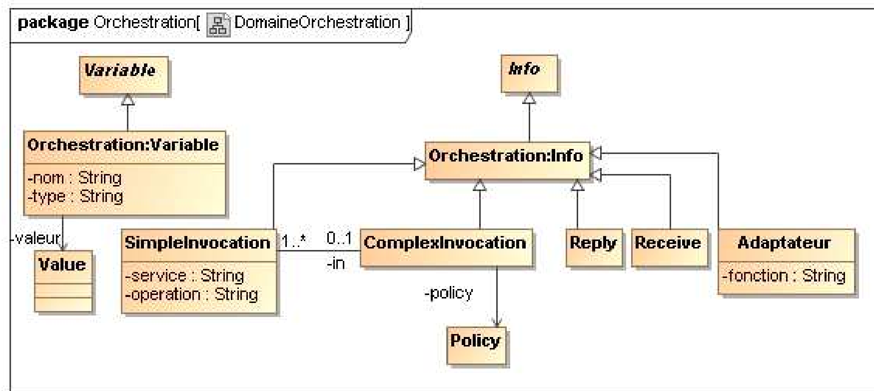



FIGURE 7.5: DOMAINE DES ORCHESTRATIONS

#### 7.2.1.1 Variables

Les variables sont typées, ont un nom et éventuellement une valeur. La valeur nous permet de définir des constantes.
Deux variables sont équivalentes si elles ont le même type et leur valeurs sont équivalentes.

#### 7.2.1.2 Informations

Les informations décrivent soit des *receive*, soit des invocations simples, soit des invocations complexes, soit des adaptateurs, soit des *reply*.

**Receive** Il existe une seule activité *receive* par orchestration et elle n'a pas de prédécesseur. Les seules variables qui peuvent être manipulées sans être affectées auparavant sont celles de l'activité *receive*.

**Invocation simple** Elle est caractérisée par le nom du service et le nom de l'opération invoquée. Elle peut également référencer une invocation complexe. Les informations correspondantes sont notées :
*invoke(Service,Operation)* ou *invoke(Service,Operation,ComplexInvocationID).*
Deux invocations simples sont considérées comme équivalentes si elles font référence au même service et à la même opération ; la présence ou non d'un identifiant vers une invocation complexe n'est pas prise en compte.

**Invocation complexe** Une invocation complexe regroupe plusieurs invocations simples à un même service et une même opération. Elle est caractérisée par le nom d'un service et d'une opération et une politique de contrôle des activités simples qui la composent. Nous avons travaillé avec deux politiques : Politique de Séparation *(SP, separation policy)* qui correspond à des invocations concurrentes des services et Politique de Synchronisation (*SCEP, Synchronous Concurrent Execution Policy*) qui correspond à l'invocation concurrente des services mais avec attente de l'ensemble des prédécesseurs.
Nous notons une invocation complexe : *complexInvoke(Service,Operation,Politique).*
Deux invocations complexes sont considérées comme équivalentes si elles sont les mêmes.

**Adaptateur** Un adaptateur a pour rôle de transformer des données. Il fait référence à une fonction qui pourra correspondre à une expression XPath ou une fonction spéciale du moteur d'exécution BPELJ par exemple. Une action faisant référence à un adaptateur a une et une seule variable en sortie.
Nous notons un adaptateur : *adapt(adapter)* où *adapter* désigne une fonction.

**Reply** Une seule activité *reply* est autorisée par orchestration. Elle ne peut pas avoir de successeurs. Elle n'a pas de variable en retour et prend une et une seule variable en entrée.

**Equivalence d'informations** A l'exclusion du cas de l'invocation simple, les équivalence d'informations correspondent à vérifier l'égalité des termes.

#### 7.2.1.3 Contraintes sur domaine

Outre les contraintes énoncées via la donnée des informations, nous contraignons le domaine avec les contraintes suivantes :

1. Une orchestration (activité composite du domaine) est déterministe.
2. Il n'y a pas d'accès en lecture à une variable qui n'a pas été affectée sauf si elle est définie comme une entrée de *receive*.
3. Une invocation simple à un couple (service,opération) apparaît au plus une fois dans l'orchestration si elle n'est pas associée à une invocation complexe. S'il est nécessaire d'utiliser plusieurs fois un même service et une même opération, alors les invocations simples doivent appartenir à une même invocation complexe. Une invocation simple a une seule variable en sortie.
4. Une invocation complexe a nécessairement le même nombre de variables d'entrées et du même type respectif que les invocations simples qui la composent. Toutes les invocations simples qui composent une invocation complexe font référence au même service et à la même opération.
5. Il existe au plus une activité *reply* par activité composite. S'il est nécessaire de retourner plusieurs données, elles doivent être encapsulées dans une structure composite. Elle ne peut pas avoir d'activité successeur.
6. Il existe une et une seule activité *receive* par activité composite. Elle est la "première" de l'activité composite au sens où toute activité simple de l'activité composite l'a comme prédécesseur même indirect ($pred^*$).
7. Les cycles dans les relations de précédences sont interdits.

### 7.2.2 Détection des ensembles de fusion

La détection des paires de fusion consiste à rechercher les activités simples qui de par les contraintes sur domaine doivent être "uniques" dans l'activité composite résultante. Ainsi les paires de fusion sont constituées pour la présence de deux activités *receive*, deux invocations sur une même opération et un même service, ou deux activités *reply*.

#### 7.2.2.1 Activités pivots

A ce jour, nous considérons comme activités pivots, exclusivement : *receive*, *reply* et les *invocations simples* et *complexes*.

#### 7.2.2.2 Détection des paires de fusion

- deux instructions *receive* forment une paire de fusion ;
- deux instructions *reply* forment une paire de fusion ;
- deux *invocations* à un même service, une même opération et référençant une même invocation complexe ne forme pas une paire de fusion. En effet, si elles référencent la même invocation complexe, c'est qu'elles ont été dîtes comme ne devant pas être unifiées. Deux *invocations* à un même service, une même opération et ne référençant pas une même invocation complexe forment potentiellement une paire de fusion. Cependant,
  - si le nombre de variable en entrée n'est pas le même *un conflit de pivot* est détecté. Ce point se justifie par le fait que la surcharge n'est pas supportée par la norme WSDL 2.0 ;
  - de même un conflit est détecté si les deux activités simples font référence à des activités composites différentes.
  - pour l'instant nous ne savons pas composer deux activités complexes, il y a donc également un conflit de pivots si nous détectons deux activités complexes faisant référence à un même service et une même opération.

Dans la figure 7.3 les ensembles de fusion sont notés *s1, s2* et *s3*.

#### 7.2.2.3 Propriétés

**Cardinalités et complexité** Nous nous trouvons dans le cadre de graphes triangulés (cf. page 50). Toutes les activités *reply* (resp. receive) forment toutes deux à deux des paires de fusion. De même les invocations sur un même service et une même opération forment soit deux à deux des paires de fusion soit il y a un conflit de pivot. Le calcul du nombre des paires de fusion est donc en $O(p^2)$ et le nombre des ensembles de fusion est donc de : $2+p$, où $p$ est le nombre d'activités d'invocations faisant référence à des couples (service,opération) différents.
La complexité du calcul des paires de fusion est donc proportionnelle aux nombres d'invocations différentes et pour chacun de ces sous-ensembles à sa cardinalité.

**Respect de la propriété 6 sur domaine relative à l'absence de paires de fusion au sein d'une activité composite** Cette propriété est respectée par les contraintes qui définissent le domaine des orchestrations : une seule activité *reply* et *receive* par orchestration, et toutes les invocations simples correspondant à un même service et une même opération appartiennent à une même activité composite ou sont uniques.

**Respect de la propriété 20 sur domaine relative à l'ordre** Le calcul des paires de fusion ne dépend pas de l'ordre des activités par construction.

### 7.2.3 Transformations et composition de transformations

Les transformations élémentaires nécessaires à la fusion d'orchestrations sont en plus des transformations définies au niveau du modèle :

1. $addPrecedenceRelation_{(pred,succ)}$ : ajoute un élément de précédence.
   $addPrecedenceRelation_{(pred,succ)}(AL, R_p, R_c) = (AL, R_p \cup pred(pred, succ), R_c)$

2. $setInCompositeActivity_{(Id_S,Id_C)}$ : ajoute à une invocation simple d'identifiant $Id_S$, la référence à une activité composite d'identifiant $Id_C$.
   $setInCompositeActivity_{(Id_S,Id_C)}(AL, R_p, R_c) = (AL', R_p, R_c)$ où
   $SimpleActivity \in LA, SimpleActivity = (Id_S, (invoke(service, operation), I, O))$.
   $ActivityInComplexActivity = (Id_S, (invoke(service, operation, Id_C), I, O))$.
   $LA' = (LA \backslash \{SimpleActivity\}) \cup \{ActivityInComplexActivity\}$ Si l'activité simple n'existe pas, elle ne fait rien.

Afin de simplifier l'écriture de la fusion, nous introduisons les deux transformations composites $replace_{(ReplacedActivity,NewActivity)}$ et $unify(Varlist) \rightarrow NewVar$ qui correspondent aux transformations élémentaires suivantes.

$$replace_{(oldActivity,newActivity)} \equiv \{substituteActivity_{(oldActivity,newActivity)}, remove_{(newActivity)}\}$$

$$unify_{\{v_1...v_n\}} \rightarrow newVar \equiv \{substituteIn_{(v_i,newVar)}, \forall i \in 1..n\}$$

où la variable $newVar$ est nouvelle et du même type que toutes les variables à unifier, et sa valeur est éventuellement la valeur d'une ou plusieurs des variables $v_i$ mais toutes doivent avoir des valeurs équivalentes.

#### 7.2.3.1 Composition

Soient
$P$, l'ensemble des transformations de la forme : $addPrecedenceRelation_{(pred,succ)}$
$R$, l'ensemble des transformations de la forme : $remove_{(activity)}$
$Svar$, l'ensemble des transformations de la forme : $substituteIn_{(oldVar,newVar)}$
$Sact$, l'ensemble des transformations de la forme : $substituteActivity_{(oldId,newId)}$
$SetIn$, l'ensemble des transformations de la forme : $setInCompositeActivity_{(Id_{Si},Id_{Ci})}$

**Ordre partiel de traitement des transformations** La composition de ces transformations commence par ordonner les transformations selon l'ordre partiel suivant : $P < Sact, R < SetIn, R < Svar$ où $A < B$ expriment que les transformations de type A devront s'appliquer avant les transformations de type B.
Les ajouts de précédences doivent s'appliquer avant d'opérer le remplacement des activités pour que même dans les précédences ajoutées, il y ait substitution des activités.

**Composition des substitutions de variables** La composition des substitutions de variables consiste à rechercher l'unificateur général (cf. §3.3, la composition dans l'exemple fil rouge). Nous rappellons ici le principe. Les substitutions $\sigma_1..\sigma_n$ sont composées en recherchant l'unificateur principal, i.e. en considérant que chaque substitution $\sigma_i = \{(v_i^1, r_i^1), ...\}$ est un problème de la forme $s_i = \{v_i^1 = r_i^1, ...\}$ où les $v_i$ et $r_i$ sont des variables et les $r_i$ n'apparaissent jamais en partie gauche de l'affectation, et en recherchant l'unificateur principal $\sigma$ du problème $\bigcup_{i=1}^{i=n} s_i$. Comme cette fois, les variables peuvent être des constantes, la composition de certaines substitutions peut échouer et lever l'erreur *conflit de transformation*.
Par exemple : $\{v1 = v3, c1 = v3, v1 = v4, c2 = v4\}$ n'a pas de solution si $c1$ et $c2$ désignent deux constantes de valeurs différentes.

Le résultat de la composition des substitutions s'il n'y a pas d'échec est donc un ensemble de substitutions.

**Composition des substitutions d'activités** Dans le cadre de la composition des orchestrations, il ne peut pas y avoir plusieurs demandes de substitutions d'une même activité. Elle ne lève donc pas d'erreur.

**Composition des références à une activité complexe** Il n'existe pas plusieurs affectations à une même invocation simple d'activités composites dans le cadre de l'algorithme présenté ici. Si ce n'était pas le cas, il faudrait vérifier que cela ne se produit pas et lever une erreur dans le cas contraire.

**Compositions des transformations et détection de circuit** La composition des ajouts de précédence permet de détecter des circuits potentiels qui peuvent être détruits par le retrait d'activités, tandis que d'autres sont ajoutés par substitutions d'activités. Par exemple, les ajouts de précédence : $\{pred(invoke1, invoke2), pred(invoke2, invoke3)\}$ ne comportent pas de circuit, mais si nous avons également les substitutions d'activités : $\{substituteActivity_{(invoke1,invoke4)}, substituteActivity_{(invoke3,invoke4)}\}$, nous obtenons les éléments de précédence suivants : $\{(invoke4, invoke2), (invoke2, invoke4)\}$, et il y a donc un circuit.
Néanmoins au niveau des seules transformations, tous les circuits ne peuvent pas être identifiés, puisque la composition des transformations est établie indépendamment des données.

#### 7.2.3.2 Propriété de la composition des transformations

**Respect de la propriété 1 sur la composition des transformations indépendante de l'ordre :** Cette propriété se démontre à la fois parce que l'application des ensembles de transformations ne dépend pas de l'ordre des activités et l'ordre entre les applications de transformations nous assure l'indépendance vis-à-vis de l'ordre des activités.
Il existe un unique unificateur général qui ne dépend pas de l'ordre des substitutions au renommage des variables près. Pour les autres substitutions, les contraintes que nous imposons nous assurent que l'application de chaque ensemble de transformations ne dépend pas de l'ordre des activités.
L'ordre d'application des ensembles de transformations assure l'indépendance de l'ordre des activités. En effet, les transformations portant sur l'ajout de précédence étant exécutées en premier, les substitutions d'activités qui ne mettent jamais en jeu les mêmes activités, porteront toutes sur les mêmes éléments de précédence. Ces deux ensembles sont les seuls qui portent sur les précédences. Pour les autres transformations, elles ne sont pas en interaction. .

### 7.2.4 Fusion d'activités simples

La fusion peut quelque soit l'ensemble de fusion être retardée à un autre tour de boucle. Dans le cas où la fusion s'applique, en fonction de la nature des activités à fusionner et des choix éventuels de l'utilisateur, le résultat de la fusion sera différent. Nous explicitons ces différents cas à présent, en soulignant pour chaque fusion : les nouvelles

activités créées, les éléments de précédence ajoutés et les transformations correspondantes.

#### 7.2.4.1 $receive^n$

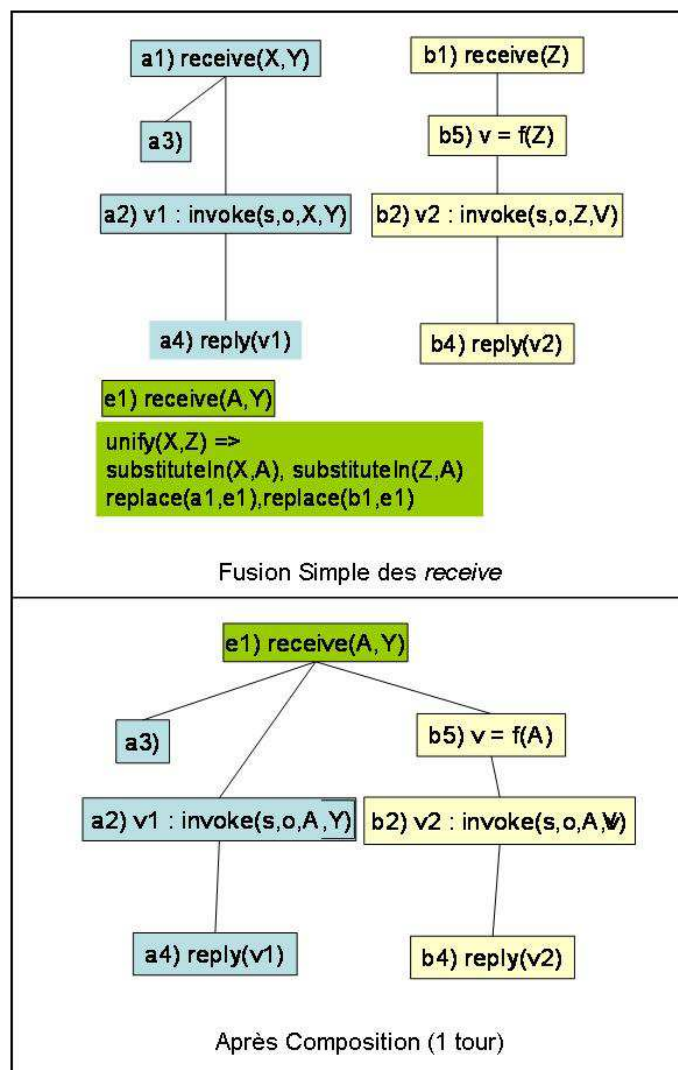
La fusion de $n$ activités *receive* consiste à créer une seule activité *receive* dans laquelle certaines variables correspondent à l'unification de variables existant dans les activités fusionnées (elles représentent la même information) et d'autres sont simplement des variables pré-existantes (elles ne représentent a priori pas les mêmes informations)(cf. figure 7.6). Nous ne cherchons pas à détecter ces unifications, elles sont données par le concepteur *(unify)* ; nous vérifions néanmoins l'unifiabilité des variables.

La fusion de $n$ activités *receive* $\{r_1, r_2, ..r_n\}$ avec $r_i = (id_i, (receive, \{v_1^i, ...v_{p_i}^i\}, \{\}))$ crée une nouvelle activité *receive* $r$ d'identifiant $id_r$. La nouvelle activité a pour variables d'entrées (i) les variables qui n'ont pas été unifiées et (ii) les nouvelles variables créées par unification.
Les transformations à appliquer sont :
(i) le remplacement des activités *receive* par la nouvelle activité *receive* :
$\{replace_{(r_1,r)}, replace_{(r_2,r)}, ..., replace_{(r_n,r)}\}$,
(ii) les substitutions des variables données par le développeur :
$\{unify(\{v_l^1, ..v_m^q\}) \rightarrow v_h, ...\}$ où toutes les variables d'un *unify* appartiennent à des orchestrations différentes.

**Résultats de la fusion d'activités *receive*** : $Activities = \{r_1, r_2, ..r_n\}$
- $todo_R = \{replace_{(r_1,r)}, replace_{(r_2,r)}, ..., replace_{(r_n,r)}\} \cup todo_s$, avec $todo_s = \{substituteIn_(x_l, v_h)...\}$ avec $x_l \in inputs(r_l), v_h \in inputs(r)$ ; $todo_s$ peut correspondre à un ensemble vide si aucune unification de variable n'a été définie ;
- $newActivities = \{r\}, r = (id_r, (receive, \{v_1, ...v_p\}, \{\}))$ avec $\forall v_i \in \{v_1, ...v_p\}$, $\exists r_k, v_i \in inputs(r_k) \vee \exists substituteIn(x, v_i) \in todo_R$
- $newR_p = \{\}$

#### 7.2.4.2 $invocation^n$

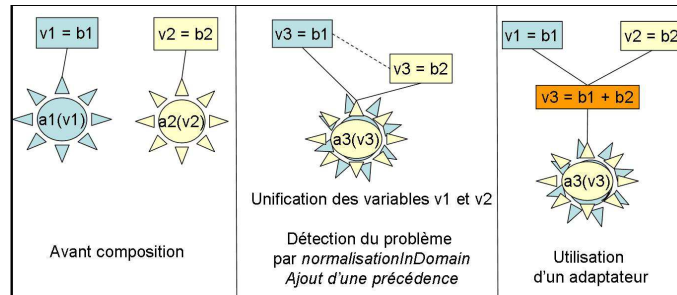
La fusion de $n$ invocations $\{i_1, i_2, ..i_n\}$ correspond (1) soit à créer une nouvelle invocation simple unificatrice et éventuellement les adaptateurs nécessaires, (2) soit à créer une invocation complexe qui coordonne les invocations simples, (3) soit à compléter une invocation complexe avec des invocations simples.
Le choix 3 s'impose en présence d'une invocation complexe.
Si entre les activités simples, l'intersection deux à deux des précédents (Pred*) avec les successeurs (Succ*) est non nulle, la seule composition autorisée est la fusion en une activité complexe respectant une politique *separate*. S'il existe déjà une activité composite appartenant à l'ensemble de fusion qui suit une autre politique, la fusion échoue. Cette contrainte permet d'éviter la création d'un circuit.
Dans les autres cas (uniquement des invocations simples sans détection d'un circuit potentiel entre elles), l'utilisateur devra choisir entre 1 et 2, et faire dans ce dernier cas le choix de la politique.




*Lors de la fusion des activités* `receive`*, les variables X et Z sont unifiées, tandis que la variable Y est laissée libre.*
*La nouvelle activité créée est ici* $e1$*. Les transformations consistent à remplacer* $a1$ *et* $b1$ *par* $e1$*, les substitutions unifient X et Z à une nouvelle variable A.*

FIGURE 7.6: FUSION ET COMPOSITION D'ACTIVITÉS *receive* PAR UNIFICATION DE VARIABLES

**1 - Fusion en une invocation simple** La figure 7.7 visualise différentes fusions possibles d'invocations simples par unification des variables ou ajout d'un adaptateur. Nous explicitons ces fusions à présent.




*La fusion des activités* `invoke` $a1$ *et* $a2$ *dans consiste simplement à remplacer ces activités par la nouvelle activité* $a3$.
*Dans la figure du milieu, la fusion est complétée par l'unification de* $v1$ *et* $v2$ *en une variable* $v3$. *Cette fusion engendre du non-déterminisme que la normalisation sur domaine lève.*
*Dans la figure de droite, la fusion crée un adaptateur qui fait la somme des variables* $v1$ *et* $v2$ *et s'interpose donc entre la nouvelle activité et les prédécesseurs des activités* $a1$ *et* $a2$.

FIGURE 7.7: EXEMPLES DE COMPOSITION D'ACTIVITÉS SIMPLES

Soient les $n$ invocations simples $i_i$ qui constituent l'ensemble de fusion :
la fusion de ces $n$ invocations simples crée une nouvelle activité simple et éventuellement un ensemble d'adaptateurs qui combinent les variables d'entrées des invocations initiales. Les variables d'entrée de la nouvelle activité correspondent donc soit à l'unification de variables d'entrées à la même place dans les invocations simples, soit à la valeur de retour d'une activité "adaptateur" créée. Les nouveaux adaptateurs précèdent la nouvelle activité simple et ont pour prédécesseurs tous les prédécesseurs de activités fusionnées.
La figure 7.8 présente un exemple de fusion introduisant un adaptateur.

**Résultats de la fusion d'activités *invoke* simples en une activité d'invocation simple** :
$Activities = \{i_1, i_2, ..i_n\} | i_i = (id_i, (invoke(service, operation), \{v_1^i, ..v_k^i\}, \{o^i\}))$
- Nouvelles activités :
  $newActivities = \{i\} \cup \bigcup_{l=1}^{l=z} \{ad_l\}$ où $0 \leq z \leq k$ [2] et
  $i = (id, invoke(service, operation), \{v_1, ..v_k\}, \{o\})$,
  $ad_l = (id_{ad_l}, (adapt(adapter_l), \{v_{l^1}, ...v_{l^m}\}, \{v_l\}))$
- Nouveaux éléments de précédence :
  $newR_p = \bigcup_{l=1}^{l=z} \{pred(id_{ad_l}, id)\} \cup \bigcup_{i=1}^{i=n} \bigcup_{id_j \in pred(R_p, id_i)} \bigcup_{l=1}^{l=z} \{pred(id_j, id_{ad_l})\}$
- $todo_R = \{unify(\{v_j^1, ...v_j^m\}), ...\} \cup \{replace_{(i_1,id)}, replace_{(i_2,id)}, ..., replace_{(i_n,id)}\}$

Le remplacement des variables par une nouvelle variable peut briser la propriété du domaine, qui interdit des accès concurrents en lecture et écriture à une même variable

2. Il y a au maximum autant d'adaptateurs que l'arité de l'invocations simple, mais il peut aussi n'y en avoir aucun.

(cf.figure 7.7, au milieu). Un accès concurrent peut être résolu par exemple en introduisant une précédence entre ces activités. Cette capacité est apportée par la normalisation sur domaine, lorsque tous les ensembles de fusion ont été résolus, des accès concurrents pouvant disparaître par unification de variables par exemple.

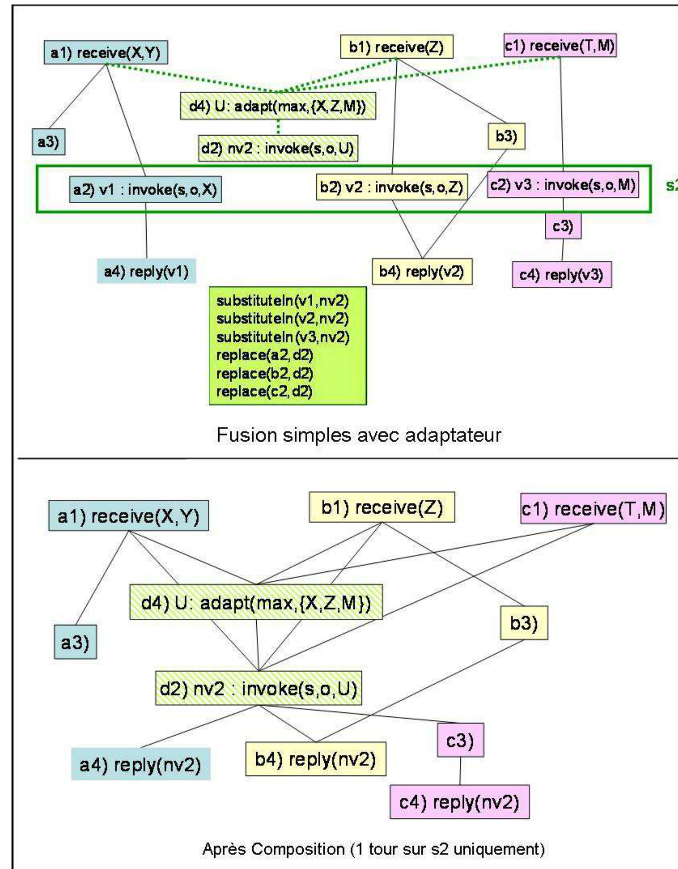
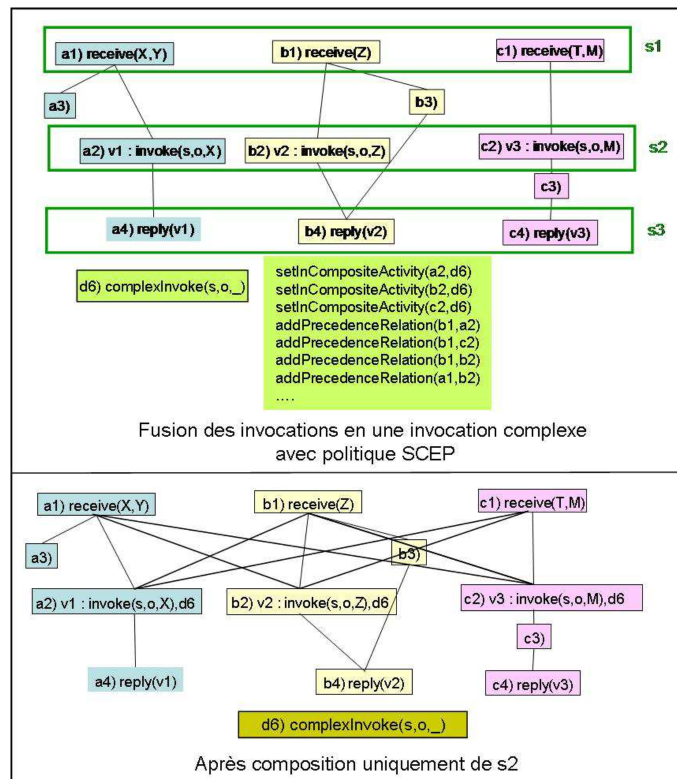



FIGURE 7.8: EXEMPLE DE FUSION D'INVOCATIONS SIMPLES AVEC INTRODUCTION D'UN ADAPTATEUR

**2 - Fusion en une invocation complexe** La fusion consiste à créer une nouvelle activité composite d'identifiant $id$ et composée des activités $id_i$.
Pour cela, les activités simples $id_i$ sont modifiées pour leur associer l'invocation complexe. Dans le cas d'une politique *separate* les relations de précédence et les variables ne sont pas modifiées. Dans le cas d'une politique de *synchronisation*, chacune des invocations simples a pour prédécesseur l'union des prédécesseurs de toutes les activités simples. Ces relations, ne mettant pas en jeu une nouvelle activité, sont gérées par transformations. La figure 7.9 visualise un exemple d'une telle composition.
Les transformations à appliquer sont alors la modification des activités simples et l'ajout de précédence.

**Résultats de la fusion d'activités *invoke* simples en une activité d'invocation complexe par Politique *Separate*** :
$Activities = \{i_1, i_2, ..i_n\}/i_i = (id_i, (invoke(service, operation), \{v_1^i, ..v_k^i\}, \{o^i\}))$




*Les activités $a2$, $b2$ et $c2$ sont considérées comme ne pouvant pas être fusionnées mais devant être synchronisées. On enregistre cette information via une activité complexe, ici $d6$. Cette activité n'est pas connectée aux graphes des activités par une relation d'ordre parce que les relations d'ordre ont été déployées directement sur les activités appartenant à l'activité complexe.*

FIGURE 7.9: EXEMPLE DE FUSION D'INVOCATIONS SIMPLES EN UNE INVOCATION COMPLEXE

- $newActivities = \{i\}\ i = (id, invokeComplexe(service, operation, SP), \{v_1..v_k\}, \{o\})$, où $\{v_1..v_k\}$ sont des variables libres
- $newR_p = \{\}$
- $todo_R = \bigcup_{i=1}^{i=n}\{setInCompositeActivity_{(id_i,id)}\}$

**Résultats de la fusion d'activités *invoke* simples en une activité d'invocation complexe par Politique de synchronisation :**
$Activities = \{i_1, i_2, ..i_n\}/i_i = (id_i, (invoke(service, operation), \{v_1^i, ..v_k^i\}, \{o^i\}))$
- $newActivities = \{i\}\ i = (id, complexInvoke(service, operation, SCEP), \{v_1..v_k\}, \{o\})$, où $\{v_1..v_k\}$ sont des variables libres
- $newR_p = \{\}$
- $todo_R = \bigcup_{i=1}^{i=n}\{setInCompositeActivity_{(id_i,id)}\} \cup \bigcup_{i=1}^{i=n}\bigcup_{id_j \in pred(R_p,id_i)}\{addPrecedenceRelation_{(id_j,id_i)}\}$

**3 - Complément d'une invocation complexe** S'il existe une invocation complexe et des invocations simples faisant référence au même service et à la même opération, les activités simples sont considérées comme composant l'activité composite. Donc, seule l'étape de création de l'activité complexe est retirée de la fusion présentée précédemment, pour le reste nous avons exactement le même comportement.

#### 7.2.4.3 $reply^n$

L'objectif de la fusion des instructions de retour $r_1..r_n$ est d'obtenir une seule valeur de retour. Il existe deux formes de fusion possibles qui sont visualisées dans la figure 7.10. Nous les explicitons à présent.

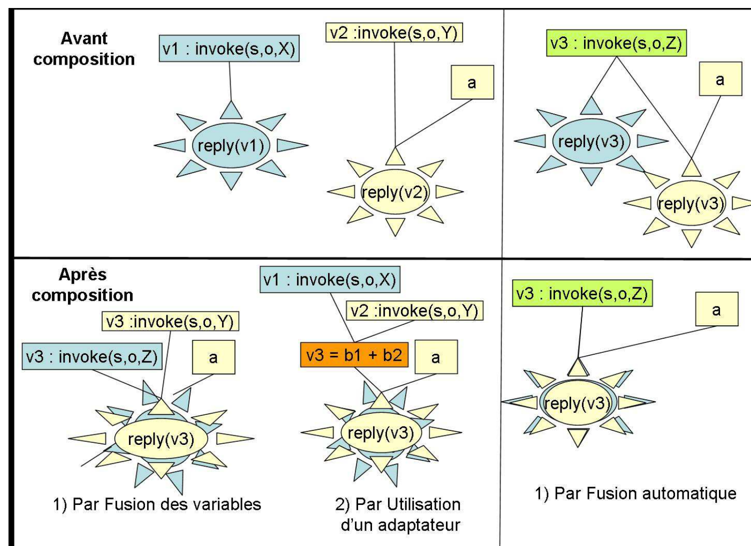



FIGURE 7.10: EXEMPLE DE COMPOSITIONS D'ACTIVITÉS *reply*

**1- par la fusion des variables en entrée** Il s'agit de créer une nouvelle activité *reply* $r$ dont la variable d'entrée est $v_r$ résultat de la fusion des variables d'entrées. Ce choix n'est possible que si les variables sont unifiables.

Par contre, cette fusion est automatique si toutes les variables ont déjà été unifiées (unification obtenue lors d'une précédente itération du processus de composition).
C'est cette forme de fusion qui a été appliquée dans les figures 7.4 et 7.8.

**Résultats de la fusion d'activités *reply* par fusion des variables :**
$Activities = \{r_1, r_2, ..r_n\} | r_i = (id_i, (reply, \{v^i\}, \{\}))$
- $r = (id, reply, \{v_r\}, \{\}), newActivities = \{r\}$
- $newR_p = \{\}$
- $todo_R = \{unify(\{v^1, ...v^n\}) \rightarrow v_r\} \cup \{replace_{(r_1,id)}, replace_{(r_2,id)}, ..., replace_{(r_n,id)}\}$

**2- par l'introduction d'un adaptateur de composition des variables d'entrées** (cf. figure 7.10 ). Dans ce cas, la variable d'entrée $v_r$ de la nouvelle activité *reply* correspond à la variable de sortie d'une activité adaptateur $ad$ créée pour gérer la composition des variables d'entrées de $r_1..r_n$, qui deviennent variables d'entrées de l'adaptateur. L'adaptateur a pour précédents l'ensemble des précédents des activités $r_1..r_n$. La nouvelle activité *reply* créée a pour précédent $ad$.

**Résultats de la fusion d'activités *reply* par composition des variables d'entrées :**
$Activities = \{r_1, r_2, ..r_n\} | r_i = (id_i, (reply, \{v^i\}, \{\}))$
- $r = (id, reply, \{v_r\}, \{\})\ ad = (id_a, (adapt(adapter), \{v^1, ..., v^n\}, \{v_r\}))$
  $newActivities = \{i, ad\}$
- $newR_p = \{pred(id_a, id)\} \cup \bigcup_{i=1}^{i=n} \bigcup_{id_j \in pred(R_p, id_i)} \bigcup_{j=1}^{j=z} \{pred(id_j, id_{ad_a})\}$
- $todo_R = \{replace_{(r_1,id)}, replace_{(r_2,id)}, ..., replace_{(r_n,id)}\}$

#### 7.2.4.4 Propriétés de la fusion d'activités simples

**Respect de la propriété 3 sur domaine relative à la non-création de circuit**
Nous démontrons la non-création de circuit par fusion en étudiant les différentes formes de fusion supportées par $mergeSimpleActivities$.
- La fusion d'activités *receive* ne peut pas créer de circuit puisque les seules modifications de la relation de précédence portent sur l'ajout de successeurs à la nouvelle activité *receive* créée, qui n'a pas de prédécesseur.
- La fusion d'activités *reply* ne peut pas créer de circuit puisque les seules modifications de la relation de précédence portent sur l'ajout de prédecesseurs à la nouvelle activité *reply* créée qui n'a pas de successeur et à la nouvelle activité adapteur éventuelle qui s'intercale entre des précédences pré-existantes.
- La fusion d'activités *invoke* modifie la relation de précédence en respectant les éléments de précédence préexistants. Or nous n'acceptons de fusionner deux activités simples que si l'intersection entre leurs précédents et successeurs est non nulle.
En conséquence, la fusion entre activités simples ne créé pas de circuit.

**Respect de la propriété 4 sur domaine relative à l'absence d'incohérence** Elle est vérifiée d'office puisque nous ne gérons pas les conditionnelles.

**Respect de la propriété 2 sur domaine relative à l'indépendance de l'ordre des activités** Cette propriété se démontre par l'indépendance de l'ordre dans la composition des transformations (cf. page 121) et le fait que chaque fusion d'activités simples

traite de manière équivalente toutes les activités ou impose l'unicité de l'activité "spéciale" comme une invocation complexe.
Notons cependant que les choix de fusion par l'utilisateur peuvent donner des rôles différents aux activités par exemple par l'introduction d'adaptateurs. L'influence de l'ordre des activités ne rentre cependant pas en ligne du compte sauf peut-être de manière subjective mais cela sort du cadre de ce travail.

### 7.2.5 Normalisation partielle

La normalisation partielle a essentiellement pour objectif dans ce domaine de détecter les circuits que l'application des transformations a pu créer.
En effet, nous avons démontré que la fusion ne crée pas de circuit. Cependant par application des transformations, il est possible de créer des circuits si nous appliquons dans une même itération plusieurs fois la fusion d'invocations simples en des invocations complexes avec politique de synchronisation. Ainsi, les contraintes relatives à l'autorisation de fusion uniquement si l'intersection entre les prédécesseurs et successeurs est nulle ne porte que sur les activités avant que les transformations ne soient effectuées. De la sorte il est possible qu'une fusion par politique *SCEP* ait introduit des précédences non visibles et que la composition des transformations ne peut pas détecter puisque l'analyse ne se fait que sur les nouvelles précédences. La normalisation partielle a alors en charge de détecter les circuits et d'interrompre le processus de composition.
Si une variable de l'activité *receive* n'est jamais utilisée en lecture avant d'être affectée, la normalisation partielle retire cette variable. Cela se produit quand une variable a été unifiée lors d'une précédente itération. Si une même variable apparaît plusieurs fois en entrée d'un *receive*, une seule occurrence est gardée.
Nous ne recherchons pas plus de points au niveau de cette étape, puisque les prochaines itérations peuvent résoudre des problèmes comme le non-déterminisme. Nous acceptons donc ces inconsistances transitoires qui seront éventuellement détectées, voir réparées lors de la normalisation sur domaine.

### 7.2.6 Normalisation sur domaine

La fonction $normaliseOnDomain_{orchestrations}$ vérifie que les contraintes énoncées par le domaine (cf. 7.2.1.3) sont vérifiées. Si une variable non affectée est accédée en lecture, une précédence est ajoutée entre l'instruction qui a besoin d'accéder à la valeur de cette variable et une instruction qui affecte cette variable et peut la précéder. S'il y a plusieurs possibilités, c'est l'utilisateur qui choisit, de même en cas de non déterminisme.
Les attentes redondantes sont retirées :
une attente $pred(X, Y)$ est redondante si $pred(X, Z) \wedge pred(Z, Y)$.

### 7.2.7 Composition d'un ensemble d'orchestrations

#### 7.2.7.1 Traitements des ensembles de fusion

L'ordre de traitement des ensembles de fusion va dépendre de l'utilisateur. En effet, si dans certains cas il est facile d'identifier les variables communes à deux orchestrations dans d'autres cas, c'est l'identification des invocations partageables qui est plus facile, entraînant du coup l'unification de certaines variables d'entrées par exemple.

De la sorte le traitement d'un ensemble de fusion peut aider à résoudre d'autres ensembles de fusion. Pour cela nous autorisons le développeur à ne pas résoudre en une itération tous les ensembles de fusion. Inversement, la résolution de plusieurs ensembles de fusion peut conduire à une activité composite invalide, par exemple en créant des circuits. Au moins un ensemble de fusion doit être résolu par itération.

#### 7.2.7.2 Respect des propriétés exigées

**Décroissance du nombre d'ensembles de fusion** La fusion d'un ensemble de fusion engendre des activités qui ne formeront plus un ensemble de fusion. En effet, la résolution des *reply* (resp. *receive*) génère un unique *reply* (resp. *receive*) et les autres activités sont retirées ; la fusion d'activités *invoke* si elle engendre une activité simple retire toutes les activités qui composaient l'espace de fusion et l'activité résultante est donc unique ; la fusion en une activité complexe assigne à toutes les activités simples l'identifiant de l'activité composite et elles ne sont donc plus fusionnables. Les adaptateurs ne sont pas des activités pivots. A chaque itération au moins un ensemble de fusion est résolu.

**Validité des activités** Nous avons montré qu'un tour de composition peut créer des circuits, mais que la normalisation partielle a en charge de les corriger ou de stopper le processus de composition. Nous n'avons pas de problèmes de cohérence puisque nous ne manipulons pas de conditions dans ce domaine.

### 7.2.8 Propriétés des compositions d'orchestrations

#### 7.2.8.1 idempotence

En composant plusieurs fois la même orchestration, le résultat est dépendant à la fois des activités qui la composent et de la constance de l'utilisateur dans ses choix de composition. Une activité composée uniquement d'une activité *receive*, *invoke* et *reply* est idempotente si les choix de fusion sont l'unification de toutes les variables de l'activité receive, la fusion en une invocation simple des invocations simples, ce qui conduit à l'unification des activités *reply*.

**idempotence** Le respect de la propriété 6 sur le domaine des orchestrations (cf. 7.2.2.3) nous permet seulement d'assurer que la composition d'activités composite dans le domaine des orchestrations est idempotente (cf.propriété 5).

#### 7.2.8.2 Préservation "partielle" des propriétés

**Il n'y a pas préservation des activités.** En effet, nous autorisons l'utilisateur à retirer des activités ou à modifier les activités résultats des fusions, de telle sorte qu'elles ne sont pas équivalentes aux activités fusionnées. Ainsi les activités *receive* ne sont pas forcément équivalentes, et le résultat de la composition n'assure donc pas que la nouvelle orchestration aura une activité *receive* ayant le même nombre de paramètres.
Par contre sur les activités d'invocation, l'activité de remplacement est à chaque fois équivalente à l'activité remplacée ; en particulier nous utilisons là l'équivalence entre les invocations simples faisant référence ou non à une activité composite.

$\Longrightarrow$ En conclusion la *préservation des activités* n'est potentiellement pas vérifiée sur la composition d'orchestrations, elle l'est sur les activités différentes de *receive* et *reply*.

**La préservation des relations de précédence est par contre vérifiée.** En effet toutes les activités préservées ont au moins les mêmes précédents et successeurs à l'équivalence près ; les seules modifications de relations sur des activités préservées sont des ajouts.

#### 7.2.8.3 Circonscription de l'absence de duplication

Si nous prenons comme hypothèse qu'il n'existe pas d'activités dupliquées dans les orchestrations à composer, nous démontrons que la composition d'orchestrations peut engendrer des activités dupliquées au sens où des activités simplement équivalentes peuvent par composition être soumises exactement aux mêmes relations. Par composition d'orchestrations, une unique activité *reply* et une unique activité *receive* sont engendrées. Par définition de $mergeSimpleActivities$, les invocations simples sont soit introduites dans une invocation complexe, soit fusionnées en une seule activité. Dans le premier cas, les activités sont soit non modifiées et donc non dupliquées, soit elles doivent respecter l'ensemble des précédences des activités de l'ensemble de fusion. Dans ces circonstances, si une activité d'invocation n'a pas de précédent autre que le *receive*, alors cette activité peut devenir équivalente dans l'orchestration résultante à une autre activité pour autant que les deux n'aient pas de successeurs. Ces deux activités étant initialement équivalentes, nous pouvons considérer qu'il n'y a pas duplication. En conséquence, l'algorithme de composition n'est pas directement mis en cause, néanmoins il apparaît important de noter que la composition d'orchestrations n'exclut pas la possibilité de créer des activités dupliquées dont il appartient à l'utilisateur de décider de la pertinence, en particulier via des adaptateurs qui peuvent être redondants et qu'à ce jour nous ne fusionnons pas.

#### 7.2.8.4 Indépendance des ordres de composition, pas des choix utilisateurs

Nous avons démontré l'indépendance de l'ordre des activités pour calculer les paires de fusion (cf. page 119) et la fusion (cf. page 126). Nous avons également démontré l'indépendance de l'application des transformations vis-à-vis de l'ordre des transformations (cf. page 121). Ces différents points nous permettent d'assurer l'indépendance de l'ordre des activités composites. Néanmoins nous avons vu que les choix utilisateurs peuvent modifier l'ordre de traitements des ensembles de fusion. Si nous supposons que les mêmes choix sont faits quelque soit l'ordre, il y a indépendance, à la nuance près que certains choix de composition seront interdits et que l'erreur est alors différente.

#### 7.2.8.5 Pas d'associativité

Dans ce domaine, de par la fonction de fusion que nous avons définie et la grande liberté dans la normalisation sur domaine, il n'y a pas associativité dans le cas général. En conséquence, nous ne garantissons pas que la composition de deux orchestrations $o1$ et $o2$ puis d'une troisième $o3$ donnera un résultat équivalent à composer $o2$ et $o3$ puis $o1$.
En particulier alors que la fusion de deux invocations et d'une activité composite est possible, nous n'autorisons pas la fusion de deux activités complexes quand bien même elles auraient la même politique. Ce point est une extension envisagée, mais non traitée

à ce jour, qui nous permettrait d'avancer vers l'associativité des compositions d'orchestrations.
De plus le fait que l'utilisateur puisse modifier les activités composites pour les rendre déterministes en ajoutant des relations de précédence peut interdire des compositions ultérieures. Nous n'avons pas de solution à ce jour.

## 7.3 Mise en œuvre et applications

La composition présentée ici a été développée en Prolog. Les interventions de l'utilisateur sont réalisées par l'assertion de faits Prolog. Cette approche nous apporte l'inférence utile à la déduction de certaines compositions et facilite la détection des paires de fusion qui sont, elles, automatiques.
Le passage de Scuffl à Prolog pour l'application décrite au paragraphe 7.1.2 a été réalisée à la main.

Un nouvel environnement ADORE plus centré sur l'évolution des orchestrations est en cours de développement [BCC+06]. Utilisé dans le projet RNTL Faros, il permet d'intégrer à une orchestration des évolutions en particulier pour supporter la mécanique de vérification de contrats. Dans cet atelier, le métamodèle d'orchestration est décrit en Ecore et des transformations de BPEL vers le métamodèle ont été implémentées en Kermeta. Puis des transformations de ce métamodèle vers Prolog et inversement nous permettent d'aborder la composition des orchestrations sur la base du code de composition déjà écrit. A terme nous envisageons cet environnement sous la forme d'un entrepôt de workflows prédéfinis répondant à des problèmes spécifiques (réception de matériaux, envoi de factures, gestion des réclamations) ou plus généraux (gestion des commandes de matériaux, suivis des clients, ...). Ces workflows pourront alors être adaptés par l'utilisation d'évolutions que nous n'avons pas présentées dans ce mémoire [MBFR08]. Il est alors possible d'utiliser conjointement plusieurs workflows généraux en détectant les points de fusions potentiels et en les composant.
Dans notre équipe, nous travaillons sur deux domaines d'applications qui mettent en jeu des assemblages de services. L'un, dit SEDUITE, est un système de diffusion d'informations au sein d'établissement scolaire. Une version est actuellement en fonction dans l'école polytechnique de Nice-Sophia Antipolis et une autre version a été installée à l'institut Clément Ader [BFCL+07]. En exploitation, ces applications exigent une forte adaptabilité pour répondre aux modifications des services, à l'extension des écoles avec l'ouverture et disparition de différents départements et activités. Cette application nous sert de démonstrateur pour le projet RNTL Faros. L'algorithme présenté pourrait être intégré dans l'année qui vient pour faciliter l'intégration de plusieurs workflows dans un même établissement.
L'autre domaine applicatif auquel nous commençons à nous intéresser porte sur la construction de workflows dans le cadre d'applications sur grilles de calcul. L'application décrite au paragraphe 7.1.2 est issue de ce domaine. Cette application utilise MOTEUR, un moteur de workflow, qui est interfacé avec l'infrastructure de production du projet EGEE. Dans ce contexte, nous pensons nous intéresser à des workflows plus importants qui permettraient de valider l'approche.
Dans les deux cas, le modèle proposé ici devra permettre d'adapter plus rapidement et efficacement les orchestrations. Ces démonstrateurs nous permettront de valoriser nos travaux par des applications réelles.

## 7.4 Conclusion et Perspectives

Nous avons proposé un algorithme de composition des orchestrations qui permet à partir d'un ensemble d'orchestrations d'obtenir une nouvelle orchestration valide. Nous avons établi cet algorithme en considérant les activités pivots de reception et de réponse, ainsi que d'invocations. Nous avons montré que l'intervention de l'utilisateur pouvait guider la composition et que cette interaction pouvait impacter l'indépendance vis-à-vis des ordres de composition. Nous avons également démontré que la préservation des activités n'était pas garantie sur les activités *receive* et *reply* : la signature du service correspondant à l'orchestration résultante n'est pas forcément "équivalente" aux signatures des orchestrations initiales, et certaines attentes pour retourner la valeur peuvent être retirées. Cette propriété n'apparaît pas utile à ce domaine. Par contre, la composition d'orchestration n'autorise pas de suppression de précédence, ce qui garantie le respect des protocoles initiaux. Ainsi il n'est pas possible dans une telle composition de modifier les autres d'appels des services, ce qui garantit partiellement le respect de leur protocole exposé [Des08].

### 7.4.1 Retour sur la composition d'orchestrations

Nous avons essentiellement porté notre attention sur l'optimisation des orchestrations résultantes et l'aide au développeur dans cette tâche. Ce faisant nous assurons la clôture de la composition et en particulier nous vérifions l'absence de circuit et le déterminisme de l'orchestration résultante. La vérification d'autres propriétés liées à l'application est envisagée d'une part sous la forme de règles Prolog, d'autres parts par transformation vers des algèbres de processus adaptées [Pou07] si leur expression est impossible ou trop difficile.
Nous avons choisi de ne pas imposer un ordre de traitement des ensembles de fusion. En suivant un ordre de parcours en profondeur des "arbres" d'activités en considérant la relation de précédence, il semble que nous aurions pu utiliser les techniques de surimposition des arbres [ALB+07]. Cependant, nous travaillons sur des DAGs puisqu'il est essentiel dans un worklow de pouvoir introduire des points de synchronisation, et la surimposition ne s'applique donc pas directement. De plus comme nous l'avons montré, la résolution des ensembles de fusion n'est pas facile. Il est important d'autoriser le programmeur à sélectionner les ensembles de fusion qu'il sait résoudre, ce qui nous permet en fonction de ses choix de réduire les possibilités de compositions et d'automatiser certaines transformations.
Ce travail commencé avec Clémentine Nemo [NBFKR07, NGBFM07] est en cours avec Sébastien Mosser [MBFR08]. C'est un des domaines qui nous paraît le plus intéressant en terme de composition parce que par nature la démarche SOA intégre la composition comme un fondement. De nombreuses perspectives sont donc en cours ou envisagées afin d'étayer la validité de cette composition en terme d'utilisabilité et de réponses aux besoins des développeurs.
L'utilisation de la réprésentation intermédiaire et l'identification des étapes de l'algorithme ainsi que des propriétés facilite assurément la construction de la composition d'orchestrations.

### 7.4.2 Perspectives

Ce travail, encore jeune, est au cœur de nos préoccupations actuelles.

#### 7.4.2.1 Modélisation et composition étendue des orchestrations

Dans le travail présenté nous avons limité la capture des langages tels que BPEL aux éléments de coordination élémentaires. La prise en compte de la gestion des erreurs en particulier des mécanismes de compensation est un des éléments qu'il nous semble essentiel d'intégrer ; la composition des gestions d'erreurs est une tâche particulièrement difficile. De même la notion d'invocation complexe est trop simple pour répondre à toutes les exigences de la composition des orchestrations.
Les possibilités de composition des activités quoique déjà complexes restent limitées. Il serait intéressant d'offrir à l'utilisateur d'autres possibilités de composition, en particulier des données. En rapprochant nos travaux de ceux sur les flots de données, nous pensons aborder ce point[MGL06], et intégrer les notions de boucles de traitement de données.
L'assemblage de services existants nécessite l'utilisation de code "glue" lorsque les structures de données consommées et produites sont différentes [KMW03]. Ces adaptations reposent dans une approche BPEL par la création de nouvelles structures, les affectations et les extensions de XPath. Pour satisfaire des besoins plus importants d'adaptation, certains langages définissent leurs propres librairies de fonctions de manipulation des structures [BGK+04]. La composition de ces fonctions nous paraît une piste prometteuse, en particulier pour la composition automatique des valeurs de retour. Ce point nous permettrait d'aborder la composition des adaptateurs en automatisant partiellement au moins leur composition.
De plus différents travaux proposent d'automatiser les compositions en prenant en compte des exigences [PTBM05] ou en utilisant la sémantique des services. Ces résultats devraient également pouvoir être appliqués pour automatiser les fusions.

#### 7.4.2.2 Atelier de composition et passage à l'échelle

La composition des orchestrations n'a pas à ce jour été abordée d'un point de vue "industriel". A cela plusieurs raisons dont peut-être la principale est le peu d'accès que nous avons à de vraies orchestrations. En effet, si SOA est aujourd'hui un thème à la mode dans notre entourage peu d'industriels utilisent vraiment des orchestrations (les compositions de service sont réalisés par l'écriture de services composites) et acceptent de les diffuser. Nous pensons que ce point sera levé dans un an lorsque plus de monde aura utilisé cette forme de programmation et que les environnements d'évaluation et de modélisation des orchestrations seront plus fiables.
L'intervention du programmeur lors de la composition d'orchestrations reste importante puisque plusieurs choix s'offrent à lui. Même si nous savons automatiser quelques compositions, une plus grande expérimentation pourrait nous permettre de proposer d'autres stratégies de composition par exemple en nous basant sur des travaux de composition automatiques menées dans le domaine des composants [Des08].

#### 7.4.2.3 Flots de données et orchestrations

Les architectures à base de web services s'appuient essentiellement aujourd'hui sur une modélisation des processus sans prise en compte des données. Or, le domaine des workflows a clairement identifié le besoin de parallélisme de données et de services (cf. 1.2.2.1) et de composition de ces données (cf. 1.2.2.2) . Dans ce contexte, une des perspectives de nos recherches porte sur l'application des techniques de transformations à la composition

conjointe des processus et des données. Nous envisageons en particulier une extension des traitements et des compositions à la gestion de tableaux de données [MD03].

#### 7.4.2.4 Traçabilité

L'intention portée par certaines formes de contrôles présentes dans BPEL peut être perdue dans la transformation vers un modèle conforme à notre métamodèle. Ainsi par exemple en BPEL on pourra utiliser aussi bien une *séquence* qu'un *lien(Link)* pour exprimer la précédence entre deux activités. Le lien peut être nommé et ainsi véhiculer de la "sémantique". Par exemple, le lien "approval-to-reply" extrait de [Rub08] exprime une relation entre deux activités. Si la transformation préserve le comportement, elle n'en perd pas moins le "commentaire" correspondant. L'intégration d'informations de traçabilité est assurément une perspective à court terme de ce travail. Ce point rejoint la problématique de la distance entre le programme et sa représentation connue dans le domaine des langages [Kli08].

*Troisième partie*

# CONCLUSION ET PERSPECTIVES

# CHAPITRE 8

# Contributions

Ce manuscrit propose une vision unifiée des différents travaux que j'ai mené autour de la composition. Mes contributions en terme de publications et de contrats se situent actuellement principalement au niveau des différentes applications. Elles sont explicitées dans l'annexe A.

L'écriture de ce mémoire m'a conduite à rechercher les éléments communs à ces travaux, en même temps qu'en tant que développeur d'algorithmes de composition je cherchais à transmettre des connaissances difficiles à énoncer en particulier les propriétés et les mécanismes de composition.

Assurément les travaux relatifs à la métacompilation ont alors inspiré le travail présenté ici. En effet, les travaux sur les langages ont défini des théories et des outils pour capturer les éléments fondateurs permettant de construire une représentation intermédiaire sur la base de laquelle les opérations de " compilation " peuvent opérer. Les grammaires attribuées ont ouvert la voie à la métacompilation en considérant (i) que l'évaluation pouvait être dirigée par des informations additionnelles, les attributs, (ii) que l'ordre des évaluations pouvait être pré-calculé indépendamment du domaine cible et (iii) que les évaluations dépendantes domaines pouvaient être capturées par des fonctions définies sur le domaine d'interprétation.

A l'instar de la problématique de la métacompilation, mon objectif a donc été de fournir à la fois les moyens au concepteur d'un algorithme de composition d'activités d'énoncer les mécanismes de la composition et également de vérifier les propriétés générales de l'algorithme résultant. Je considère donc que mes contributions se situent sur les points suivants.

**Représentation intermédiaire inductive support à la composition d'activités**
Il existe plusieurs représentations des activités en fonction des applications : algèbre de processus, langage BPEL, notations graphiques. Les algorithmes de composition, quand ils existent, dépendent de ces représentations. Ma première contribution est dans *la définition de la représentation intermédiaire inductive* par un métamodèle qui est déterminé par l'objectif de composition. Elle n'est pas définie sur la base des opérateurs, mais les relations entre les activités.

**Interprétation libre et sur domaine en support à la métacomposition d'activités** Les compositions dépendent fortement du domaine cible. Pour prendre en compte cette dépendance, le raffinement des métamodèles et la notion d'interprétation permettent de capturer les contraintes du domaine applicatif. Cette *approche mixte mêlant métamodélisation, métacomposition et interprétation* constitue une autre de nos contributions. En effet, en utilisant une approche par interprétation, nous ne restreignons pas les domaines d'étude à notre modélisation. Par exemple la vérification des contraintes sur domaine lors de la normalisation partielle ou sur domaine, peut se faire dans un autre espace technologique comme par exemple les algèbres de processus [Pou07].

**Propriétés de la composition par respect des contraintes sur les interprétations** Les fonctions qui définissent une interprétation doivent respecter des propriétés assurant la terminaison de l'algorithme de composition, l'absence de circuit et la clôture de la composition. D'autres propriétés peuvent être exigées pour assurer par exemple l'associativité des compositions ou la préservation des activités, . . . La *détermination de ces propriétés et leur impact sur l'algorithme de composition* et les fonctions sur domaine constituent également une contribution. En particulier ces contraintes assurent la rectitude des graphes d'activités produits. L'explicitation des propriétés recherchées et leur impact sur l'interprétation devrait faciliter la tâche du concepteur d'algorithmes de composition. Nous considérons qu'un graphe d'activités est correct, s'il respecte les contraintes énoncées par son domaine. L'algorithme de composition que nous avons défini (interprétation libre) assure (modulo le respect des propriétés exigées) que le résultat d'une composition est bien dans le domaine des comportements attendus. Nous avons établi dans la construction même de l'algorithme les éléments pouvant occasionner des erreurs lors des compositions. Cette identification permet de bien cerner la cause des conflits lors des compositions. Cependant dans le cas d'erreur détectée au niveau du domaine (donc l'interprétation sur domaine a créé un comportement invalide), il reste difficile de localiser la cause de l'erreur.

**Approche méthodologique de la composition d'activités** De notre point de vue la circonscription de la composition d'activités avec le formalisme proposé est également une contribution méthodologique de définition des algorithmes de composition de activités, nous parlons de **métacomposition**. Ce point a pu être vérifié sur les deux applications clefs. Il nous reste néanmoins à étayer ce travail par d'autres compositions directement guidées par ce résultat pour valider cette avancée. A priori nous pouvons utiliser le même algorithme pour exprimer la composition d'aspects selon le langage Aspect-J. Outre les deux domaines utilisés en exemple, je me suis également intéressée à la composition d'activités dans le cadre de l'intégration de services dans le cadre de la thèse d'Olivier Nano[Nan04] et à la gestion de dépendances au sein d'un framework dans le cadre de la thèse de Pascal Rapicault [Rap02]. Même si je n'ai pas encore démontré que ces différents travaux pouvaient être capturés par MM4AC, c'est une perspective à très court terme. Enfin la thèse de Clémentine Nemo sur la composition de patterns[NBFR08] et celle de Sébastien Mosser sur la composition d'évolutions de workflows [MBFR08] devraient dans les années à venir aider à l'amélioration de cette approche et les perspectives qui suivent sont nombreuses.

# CHAPITRE 9

# Perspectives

Nous avons développé dans ce manuscrit une approche par métacomposition de graphes d'activités. Nous l'avons appliqué à deux domaines d'étude : les interactions entre composants hétérogènes et les orchestrations. Nous pensons que les avancées en matière de structuration de connaissances par les objets, les composants, les services, les ontologies sont aujourd'hui une base bien fondée de représentation de nos connaissances que les machines sont amenées à exploiter. La représentation des activités est encore aujourd'hui fortement ancrée dans une approche code qui est la manière, pour l'heure, la plus naturelle d'exprimer ce que nous attendons de la machine ou des hommes lorsque nous représentons des workflows humains. Cependant tout comme la représentation des structures, la représentation du comportement conditionne nos capacités d'analyse et de raisonnement des systèmes. Aussi différents formalismes émergent-ils pour capturer cette information en fonction des objectifs sous la forme de diagrammes de séquences, d'activités, d'algèbres, d'aspects. Nous nous situons dans cette mouvance en proposant une représentation intermédiaire qui vise à aider les compositions de ces activités. Notre proposition est prospective. En l'état elle présente à la fois des limites et ouvre de notre point de vue la voie à plusieurs travaux de recherche.
Nous précisons dans chacune des sous-parties qui suivent les recherches qui nous semblent prioritaires et sur lesquelles nous comptons travailler à court terme.

## 9.1 Représentation intermédiaire

Contrairement aux approches sur les réseaux de Petri qui sont basées sur des états, nous avons défendu une approche basée événements, plus simple que les approches basées sur les algèbres de processus telles que le pi-calcul qui intègrent les notions de canaux et explicitent ainsi le temps de circulation des événements. Notre objectif étant limité à la composition nous simplifions l'expression d'un workflow : la présence d'un ordre entre deux activités explicite que la fin d'une activité (événement de fin) autorise l'activité suivante à s'exécuter. Cette approche a libéré les activités des relations qui les unissent ce qui nous permet de calculer les compositions sur la base de graphes. Cependant, le passé acquis autour des algèbres de processus est accessible par transformations. Ainsi nous défendons une représentation intermédiaire pour la composition sans perdre les travaux

antérieurs. Une étude plus poussée des interrelations entre ces représentations en particulier en terme d'incrémentalité dans les compositions est une de nos perspectives.

**Gestion de l'erreur** Nous avons défendu l'idée que les activités composites pouvaient être appréhendées comme des graphes d'activités simples dont les arêtes capturent les informations de précédence et de conditions. Nous envisageons d'ajouter une relation (arêtes dans le graphe) qui capture la gestion des exceptions. A chaque activité peuvent alors être associée plusieurs gestions des exceptions, que les algorithmes de composition ont à charge de composer.

**Autres relations entre activités** La relation de précédence stipule qu'une activité ne peut être exécutée que lorsque toutes les activités qui la précèdent ont été exécutées. Nous définissons ainsi des points de synchronisation. Cependant, il peut être nécessaire d'exprimer qu'une activité commence son exécution dès qu'une des activités qui la précède a terminé son exécution. Cela nous ramène à la notion de bloc ou de terme dans [BCHS07], avec en particulier l'opérateur $\boxplus$ qui permet de considérer un ensemble d'activités comme un bloc dont l'exécution est complète dès qu'une des activités qui le compose a terminé son exécution. L'introduction de cette capacité sous la forme de bloc est une de nos perspectives.

### Comportements itératifs

Nous n'avons pas pris en charge les boucles parce que la sémantique d'exécution que nous avons associée aux activités et la relation de précédence ne peuvent pas correspondre directement à des activités qui seraient répétées. De plus la prise en compte des boucles dans l'application des transformations n'est pas facile à cerner et peut impliquer des transformations secondaires [KHJ06]. Cependant elles sont un élément important de représentation des activités et nous les envisageons actuellement selon plusieurs points de vue.

**Dépliage** Une vision des boucles dans les workflows est de considérer les boucles comme de multiples activités. Dans ce contexte, les activités répétées (en séquence ou de manière asynchrone) sont explicitement représentées. Cette approche suppose une connaissance statique du nombre d'itérations et augmente la complexité des algorithmes en nous faisant perdre l'information sur l'itération.

**Blocs d'activités** Une autre approche que nous trouvons également dans la gestion des flots de données est de considérer les boucles comme des blocs d'activités. Dans le cadre de l'application des compositions d'orchestrations, cette approche est partiellement choisie, en considérant une activité complexe comme une boîte noire dont la politique d'exécution peut par exemple supporter des boucles. A ce jour, cette solution ne permet pas de supporter des compositions très évoluées et surtout ne tient pas compte des données manipulées.

**Gestion ensembliste** La solution que nous privilégions aujourd'hui est de considérer les boucles comme des manipulations d'ensembles de données par analogie avec un moteur d'exécution de flots de données. Ce travail est en cours. Nous trouvons des approches

équivalentes dans les calculs sur grilles avec l'introduction d'opérateurs de composition des données [MGL06].

### 9.1.1 Perspective prédominante : représentation intermédiaire incluant la gestion des flux de données

Au travers de la thèse de Sébastien Mosser nous portons aujourd'hui notre attention sur une représentation étendue des activités composites qui nous permettrait de gérer des flux de données tels que ceux présents dans le cadre des applications en imageries médicales sur grilles de calcul. Notre intérêt pour ce travail s'explique par (i) un cadre applicatif réel dans lequel notre contribution devrait permettre à terme un accès plus facile aux grilles de calculs aux spécialistes en imageries médicales et aux médecins, (ii) la synergie de deux domaines de recherche - la composition de flots de données et la composition de flots de contrôles- qui nous permet d'explorer de nouvelles pistes a priori peu abordées aujourd'hui en terme d'expression et de contrôle des évolutions.

## 9.2 Détermination des ensembles de fusion

L'algorithme de composition est basé sur la détermination de paires de fusions. La détermination des paires est dépendante domaine et porte sur l'ensemble des activités. Les paires sont considérées comme indépendantes les unes des autres, ce qui nous permet de construire des cliques. Les ensembles de fusion sont donc construits indépendamment de la sémantique des paires elles-même.

**Activités pivots et matching** Dans les domaines que nous avons étudiés nous nous sommes intéressés à des paires de fusion dont la détermination est basée sur une identification syntaxique : nom des activités, signatures, ... Dans le domaine des composants et des services, nous trouvons les notions de substituabilité qui intègrent d'autres éléments comme le comportement. Par exemple, si une activité fait référence à une activité composite, la détection du matching pourra inclure la reconnaissance de comportements compatibles. Dans [NSC+07], les auteurs proposent une composition des diagrammes d'états qui repose sur une étape de "matching" des états utilisant des heuristiques mettant en jeu le nom des états (typographie et linguistique) et le comportement par analyse des transitions entre états. De telles extensions devraient compléter les travaux que nous avons menés sur la composition des orchestrations.

**Des paires d'activités à des paires d'ensembles d'activités** Dans les travaux de Klein et autres, le résultat du "matching" entre un point de coupe énoncé sous la forme d'un modèle de scénario (que nous assimilons à une activité composite) et un scénario est un ensemble d'activités composites [KHJ06]. L'objectif est différent puisque dans ce contexte les deux activités composites ne jouent pas le même rôle : un aspect énoncé sous la forme de deux scénarios (l'un modèle, l'autre advice) doit être intégré dans un scénario initial, par remplacement des activités qui "matchent" le modèle de scenario par l'advice. Cependant, la démarche est vraiment intéressante et pose la question d'une composition dans laquelle les ensembles de fusion correspondent à des ensembles d'activités. Ce point fait assurément partie d'une de nos perspectives. Il a également été évoqué par Didonet dans [FV07].

### 9.2.1 Perspective souhaitée : application à la composition de scénarios

Nous aimerions reprendre les travaux sur la composition de scenarios présentés dans[KFJ07] pour étendre l'algorithme de composition que nous avons proposé et intégrer cette approche de la composition. Notre intéret est de vérifier la validité de la métacomposition en l'appliquant à un autre domaine en particulier en vérifiant ainsi le respect ou non de certaines propriétés de la composition.
De plus les travaux en cours avec Sébastien Mosser proposant une approche de l'évolution des graphes d'activités basée sur l'explicitation d'un point de jonction et l'explicitation de la place des prédécesseurs et successeurs, cette intégration faciliterait la comparaison des approches. Enfin, la composition de scenarios se base sur l'ordonnancement des ensembles de fusions, tandis que nous proposons d'ordonner les transformations. Il serait intéressant de vérifier que le même résultat peut être obtenu par composition des transformations.
Nous ne disposons pas, pour l'instant, des moyens d'aborder cette application qui pourtant nous permettrait d'impacter clairement le monde des modèles et de porter notre travail au niveau des standards.

## 9.3 Validité des compositions d'activités

La composition des activités peut entraîner des modifications des qualités de services[Cas05]. Différents travaux s'intéressent à la composition des contrats [COR06, BCC+06]. Une de nos perspectives est de prendre en charge dans la composition des activités la composition des contrats et éventuellement de rejeter certaines compositions qui ne satisfont pas les exigences [JFBF06].

### 9.3.1 Perspective prédominante : Associer contrats et composition

Vérifier que les compositions d'activités respectent bien certains contrats est particulièrement utile lorsque l'on veut faire évoluer des applications sous contraintes en s'assurant que ces contraintes sont toujours vérifiées. Une discussion est en cours avec la société Maat-G pour la mise en place d'une bourse Cifre qui nous permettrait d'aborder cette problématique dans le cadre d'applications en imagerie médicale sur grilles de calculs devant respecter des propriétés telles que la sécurité des accès aux données.
Nous considérons cette perpsective comme prédominante car elle permet de pousser plus loin les limites d'une composition indépendante des applications (complexité des infrastructures, nature des compositions, taille des applications) tout en abordant un des problèmes fondamentaux des nouvelles applications, la qualité des services offerts.

## 9.4 Transformations

**Décomposition des transformations** Nous avons suivi une démarche où n activités composites sont composées. Défaire une composition revient à recalculer la composition en ôtant l'activité composite qui n'intervient plus dans la composition. Ce choix n'est possible que parce que d'autres actions n'ont pas modifié l'activité résultante après composition. Dans le contexte, du développement d'une application par incréments (stepwise), il est nécessaire de pouvoir défaire une composition sans perdre les éléments qui ne lui sont pas corrélés [AL06]. Nous étudions cet aspect de la composition au travers

du travail de Clémentine Nemo sur la multi-application de patterns, l'idempotence des transformations et le mode *replay* [NBFR08].

**Composition des transformations** Nous avons établi "à la main" l'ordre des transformations. Or, différents travaux en ingénierie des modèles (IDM) abordent la problématique d'une composition automatique des transformations qui assurent le déterminisme de la solution quelque soit l'ordre d'application des règles ou détecte un problème [Ber03, MKR06]. La métacomposition a donc beaucoup à gagner de ces travaux et y contribue en identifiant les propriétés recherchées pour la composition des transformations.

### 9.4.1 Perspective prédominante : Composition de transformations, idempotence et rejeux

Via la thèse de Clémentine Nemo réalisée en co-direction avec la société DCNS, nous nous intéressons à maintenir "valide" l'application de "patterns" sur des assemblages de composants. Dans le cadre de cette thèse nous considérons l'expression d'un pattern comme, entre autre, une transformation qui vise à modifier le modèle initial en un modèle respectant les contraintes portées par le pattern. Pour cela, nous avons défini la notion de transformation "conformante" qui a entre autre propriété l'idempotence et la préservation de la conformité des modèles. Ainsi l'application d'une transformation conformante n'a aucune action si le modèle respecte les contraintes ciblées par cette transformation.
Ces recherches sur le contrôle et la composition de transformations d'assemblage de composants nous conduisent à aborder les problèmes de recherche du point fixe (comment assurer l'existence d'une solution lors de l'application de séquences de transformations conformantes), d'ordonnancement des transformations (comment gérer l'application de plusieurs transformations non ordonnées ?), l'incrémentalité dans les compositions (seuls les éléments nécessaires sont modifiés par application de transformations conformantes), la place des actions utilisateurs dans un ensemble de transformations.
Chacun de ces points est nécessaire au "passage à l'échelle" de la composition des activités à la fois pour le nombre de compositions à opérer, le nombre des activités à gérer et le réalisme des applications traitées.

## 9.5 Auto-organisation

Dans le monde des services et des composants se développent des outils permettant la découverte dynamique de services et de composants. Ces travaux reposent sur l'association aux services/composants de métainformations, souvent via l'usage d'ontologies [PSSN03]. Ainsi les assemblages se recomposent en fonction d'objectifs fonctionnels [Des08] pour sélectionner les services les mieux adaptés ou en cas d'échec pour substituer de nouveaux services répondant mieux aux exigences utilisateur [BW03]. En fonction des approches, il s'agit donc de prendre en compte soit de l'adaptation par l'utilisateur soit de l'auto-configuration, les auteurs parlent alors d'auto-organisation [PB05].
Dans tous les cas notre intuition est que ce n'est pas un composant qui va venir se composer avec un assemblage existant mais des assemblages qui se composent. Ainsi, à plus long terme, nous considérons que nos travaux ouvrent la voie à de nouvelles approches de l'évolution des systèmes.

# ANNEXE A

# Liste sélective de résultats et d'encadrements

Cette annexe présente par thèmes une liste sélective de publications et d'encadrements.
Une liste complète de publications est disponible sur la page web : http ://www.polytech.unice.fr/ blay.

## A.1 Modélisation de la composition d'activités

Cette partie de ce manuscrit n'a pas été publiée en tant que telle à la fois pour des raisons de temps et de complexité étant donné la longueur de cette partie. Ce travail de diffusion reste donc à faire.

### Publications et contrats

Bien que non directement associés à la modélisation des activités telle que présentée dans le manuscrit, nous considérons que les travaux suivants plus orientés applications sont relatifs à cette contribution.

- Clémentine Nemo, Mireille Blay-Fornarino, and Michel Riveill. Multi-Applications de Patterns - Formalisation. In *4ième Journée sur l'Ingénierie Dirigée par les Modèles (IDM'08)*, Mulhouse (France), June 2008.[NBFR08]
- Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles : au-delà du MDA*. Hermes-Lavoisier, Cachan, France, February 2006.[FEBF06]
- Mireille Blay-Fornarino and Paul Franchi-Zannettacci. *Au delà du MDA : l'Ingénierie Dirigée par les Modèles*, chapter "Les Langages et l'IDM". Hermès, 2006.[BFFZ06]
- Mireille Blay-Fornarino, Paul Franchi, and Olivier Nano. Vers une sémantique " plug-in " pour les modèles. In Sébastien Gérard, Jean-Marie Favre, Pierre-Alain Muller, Ivar Austvoll, and Xavier Blanc, editors, *IDM'05*, Paris, june 2005. [BFFN05]
- Olivier Nano and Mireille Blay-Fornarino. Annotations et transformations de modèles pour l'intégration de services. *RSTI - Série L'Objet (RSTI-Objet)*, 10(2-3) :175–188, 2004. [NBF04]

Associé au RNTL Faros, nous considérons que le livrable suivant, dont j'étais responsable du chapitre *composition*, est également une contribution à cette thématique.

- Mireille Blay-Fornarino, Pierre Combes, Laurence Duchien, Tristan Glatard, Philippe Lahire, Stéphane Lavirotte, Clémentine Nemo, Audrey Occello, Renaud Pawlak, Anne-

Marie Pinna-Déry, Lionel Seinturier, and Jean-Yves Tigli. Chapitre Développement par Composition dans Livrable Faros : État de l'art sur la contractualisation et composition. Technical Report F.1.1, RNTL Faros, October 2006 [BFCD+06]

### Encadrements

#### Thèses soutenues et en cours

Bien que non directement liées au contenu du chapitre relatif à la modélisation de la composition d'activités, deux travaux de thèses inspirent ce chapitre.

1. De 2001 à 2004, j'ai encadré la thèse de Olivier Nano (directeur M. Riveill) [Nan04] dans laquelle nous avons exploré l'intégration des services dans les applications à base de composants. Ce travail a mis en relief à la fois la nécessité d'une représentation intermédiaire adéquate.
2. La thèse de de Clémentine Nemo commencée en 2006 (directeur M. Riveill) aborde le problème de l'idempotence lors de l'application multiple de transformations. Ce travail inspire fortement mes recherches sur les compositions de transformations.

#### Masters

En amont de ces thèses, les masters suivants nous ont permis d'investiguer de nouvelles voies de recherche.
- En 2000, j'ai encadré Laurent Bussard sur son stage de DEA relativement à la composition de services Corba[Bus00].
- Ce travail a été poursuivi en 2001 par le stage de master de Olivier Nano.[Nan01]

## A.2 Composition d'interactions

### Publications

- M. Blay-Fornarino, A. Charfi, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Software interaction. *Journal of Object Technology*, 3(10) :161–180, z 2004.[BFCE+04]
- Anis Charfi, Michel Riveill, Mireille Blay-Fornarino, and Anne-Marie Pinna-Déry. Transparent and Dynamic Aspect Composition. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, Bonn, Germany, March 2006. [CRBFPD06]
- Daniel Cheung Foo Woo, Mireille Blay-Fornarino, Jean-Yves Tigli, Anne-Marie Pinna-Déry, David Emsellem, and Michel Riveill. Langage d'aspects pour la composition dynamique de composants embarqués. In Pierre Cointe and Lionel Seinturier, editors, *JFDLPA'05*, Lille, France, sep 2005.[CFWBFT+05]
- Mireille Blay-Fornarino, David Emsellem, Anne-Marie Pinna-Dery, and Michel Riveill. Un service d'interactions : principes et implémentation. *Technique et science informatiques RSTI série TSI (RSTI-TSI)*, 23(2) :175–204, 2004. [BFEPDR04]
- Anne-Marie Dery, Mireille Blay-Fornarino, Borice Arcier, Léonard Mule, and Sabine Moisan. Distributed access knowledge-based system : Reified interaction service for trace and control. In *DOA*, pages 76–84, 2001.[DBFA+01]
- Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna. A reflective model for first class dependencies. In *Proceedings of OOPSLA '95 (International Conference*

*on Object-Oriented Programming Systems, Languages and Applications)*, pages 265–280. ACM, oct 1995 [DBFP95].

Les travaux autour des interactions ont fait l'objet de différents contrats de recherches et d'un dépôt à l'APP.

– La spécification et la mise en œuvre du service d'interactions permettant d'établir dynamiquement des interactions entre des composants Corba/java et Corba/C++, puis en RMI, et enfin aux EJBs. est diffusé depuis le site Web du projet (http ://rainbow.essi.fr) et a été déposé à l'APP. Ce prototype a été soutenu par microsoft research en 2003 et 2004.
– Suite à la thèse de Pascal Rapicault, nous avons défini et implémenté le plugin Eclipse Babylone qui a fait l'objet d'un IBM eclipse Innovation Grant en 2003. A partir d'un framework de " modèles ", le concepteur de framework précise une métareprésentation du framework qu'il développe et en particulier les dépendances entre les " concepts " et les classes fournis. A partir de cette information, l'environnement de développement relatif au framework est alors construit automatiquement. L'intégration dans Eclipse nous permet de contrôler le développement de classes et de fichiers XML. L'utilisateur final est alors guidé dans la création, le nommage, la destruction des éléments de son application. Il peut également vérifier la consistance du système créé.
– 2001-2003, Participation au projet ARCAD (RNTL exploratoire) qui avait pour objectif d'étendre la plate-forme " logiciel libre " ObjectWeb afin de la rendre adaptable et extensible.
– 2002-2003, Participation au projet ASPECT (RNTL précompétitif) qui regroupait deux PME (Prologue software et Memsoft) et l'Université de Nice et qui avait pour objectif de définir un modèle de composants adaptables pour un mode de fonctionnement de type ASP (location de logiciel). De ce travail ont été issus les travaux sur la composition d'IHM aujourd'hui conduits par Anne-Marie Pinna-Dery.
– 2002-2003, Participation au projet IMPACT (RNTL plateforme,) qui regroupait de nombreux partenaires impliqués dans la mise en IJuvre d'intergiciel libre. Ma participation concernait le service d'interactions.

## Encadrements

### Thèses

– Avec Stéphane Ducasse de septembre 1993 à janvier 1997 (directeur Paul Franchi-Zannettacci), nous avons exploré la séparation des préoccupations dans le cadre des objets. Les aspects n'existaient pas encore.[Duc97].
– De 1999 à 2002, j'ai co-encadré avec J.P. Rigault, la thèse de Pascal Rapicault soutenue en mai 2002. Avec ce travail, les interactions ont été étendues et appliquées à la définition des frameworks[Rap02].

### Masters

– En 2006, j'ai co-encadré par Anne-Marie Pinna, Sébastien Bianco sur la vérification de l'adaptation dynamique des assemblages de composants. Ce travail nous permis d'aborder l'expression des contraintes entre interactions[Bia06].
– En 2002, pendant son DEA que j'ai encadré, Audrey Occello a amorcé l'étude relative à la sûreté des assemblages de composants [Occ02].

## A.3 Composition d'orchestrations

### Publications

- Sébastien Mosser, Mireille Blay-Fornarino, and Michel Riveill. Web Services Orchestration Evolution : A Merge Process For Behavioral Evolution. In *2nd European Conference on Software Architecture (ECSA'08)*, Paphos, Cyprus, September 2008. Springer LNCS.[MBFR08]
- Sébastien Mosser, Franck Chauvel, Mireille Blay-Fornarino, and Michel Riveill. Web Service Composition : Mashups Driven Orchestration Definition. In *International Conference on Itelligent Agents, Web Technologies and Internet Commerce (IAWTIC'08)*, Vienna, Austria, December 2008. IEEE Computer Society.[MCBFR08]
- Clémentine Nemo, Tristan Glatard, Mireille Blay-Fornarino, and Johan Montagnat. Merging overlapping orchestrations : an application to the Bronze Standard medical application. In *International Conference on Services Computing (SCC 2007)*, Salt Lake City, Utah, USA, July 2007. IEEE Computer Engineering.[NGBFM07]
- Clémentine Nemo, Mireille Blay-Fornarino, Günter Kniesel, and Michel Riveill. SEMANTIC ORCHESTRATIONS MERGING - Towards Composition of Overlapping Orchestrations. In Joaquim Filipe, editor, *9th International Conference on Enterprise Information Systems (ICEIS'2007)*, Funchal, Madeira, June 2007. [NBFKR07]
- Mireille Blay-Fornarino, Vincent Hourdin, Cédric Joffroy, Stéphane Lavirotte, Sébastien Mosser, Anne-Marie Pinna-Déry, Philippe Renevier, Michel Riveill, and Jean-Yves Tigli. Architecture pour l'adaptation de Systèmes d'Information Interactifs Orientés Services. *Revues des Sciences et Technologies de l'Information (RSTI)*, pages 93–118, December 2007. [BFHJ+07]

J'ai participé sur ce thème aux contrats suivants.

- 2006-2009, Projet RNTL Faros. Le projet FAROS a pour objectif de définir un environnement de composition pour la construction fiable d'architectures orientées services. Il complète les travaux sur l'intégration d'applications par la prise en compte d'éléments contractuels permettant une composition cohérente de services ainsi que par la définition d'une méthodologie permettant de rendre reproductibles les procédés d'intégration de contrats depuis des modèles métiers jusqu'à leur projection vers des plates-formes d'exécution. C'est sur ce dernier point que porte plus précisément ma contribution. Nous abordons actuellement l'intégration de contrats dans l'application SEDUITE que nous avons développé en interne et qui supporte aujourd'hui le système de diffusion des informations relatives à notre école sur des écrans. Ce système met en jeu des services Web et des composants WComp pour la partie multi terminaux qui est en cours de développement. Nous nous intéressons aux techniques à mettre en œuvre pour injecter les contrats dans ces plates-formes.
- 2004-2006, j'ai participé au Projet PREDIT MobiVIP principalement sur la mise en œuvre d'un système d'information multimodal basé sur la composition de services.

### Encadrements

#### Thèse en cours

#### Masters

- La composition d'orchestration a été amorcée par le travail de master de Clémentine Nemo [Nem06]

- L'approche dynamique de l'évolution d'une orchestration de web services a été étudié via le stage de deuxième année de Sébastien Mosser puis le projet de fin d'étude [JMBF07].
- Le passage du métamodèle d'activités à BPEL a été étudié par Douae Hammouche pendant son master[Ham08].

# TABLE DES FIGURES

1.1 Contexte de la composition des activités . . . . . . . . . . . . . . . . . . . . . . 6
1.2 Composition d'activités : point de vue général . . . . . . . . . . . . . . . . . . 20

2.1 Notre modélisation du concept d'activité : le métamodèle MM4CA . . . . . . 31
2.2 Exemple d'activités en UML 2.0 (extrait de [Boc03]) . . . . . . . . . . . . . . 31
2.3 Exemple de [Boc03] modélisé conformément au métamodèle MM4CA . . . . 31
2.4 cycle de vie d'une activité composite . . . . . . . . . . . . . . . . . . . . . . . 32
2.5 cycle de vie d'une activité simple . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.6 Domaine de l'exemple fil rouge . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.7 Représentation graphique des activités présentées dans l'exemple Fil Rouge de la page 36. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
2.8 Exemple de normalisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

3.1 Exemple de composition d'activités composites dans le domaine Fil Rouge . 49
3.2 Exemple de fusion d'activités simples (a13,a22,a32) . . . . . . . . . . . . . . 57
3.3 Résultat de la Composition avant normalisation partielle . . . . . . . . . . . 62
3.4 Résultat de la Composition après normalisation partielle . . . . . . . . . . . 62

6.1 Graphe d'interactions pour l'application Agenda . . . . . . . . . . . . . . . . 84
6.2 Schémas d'interactions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.3 Domaine des Interactions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
6.4 ISL, Interactions et Formalisation . . . . . . . . . . . . . . . . . . . . . . . . . 87
6.5 Exemples de composition d'interactions . . . . . . . . . . . . . . . . . . . . . . 90
6.6 Exemple de composition d'activités autour de *proceed* . . . . . . . . . . . . . 91
6.7 Exemple de composition d'activités autour de *absorb* . . . . . . . . . . . . . 91
6.8 Composition d'activités "pivots" et conditionnelles . . . . . . . . . . . . . . . . 91
6.9 Schémas, règles d'interactions et activités composites . . . . . . . . . . . . . 96
6.10 grammaire ISL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
6.11 Visualisation partielle du graphe d'interactions . . . . . . . . . . . . . . . . . 104
6.12 Comparaison des compositions par l'exemple entre AspectJ et ISL (*proceed*) 107

6.13 Comparaison des compositions par l'exemple entre AspectJ et ISL (*delegate* et ordre de composition) . . . 107

7.1 Orchestrations CM (gauche), PFM (milieu) et composée (droite). . . . 115
7.2 Détection des paires de fusion et détermination des opérations de fusion . . 115
7.3 Exemple de compositions d'orchestrations : de la détection des ensembles à la composition des transformations. . . . 116
7.4 Exemple de compositions d'orchestrations : applications des transformations et normalisation sur domaine . . . 117
7.5 Domaine des orchestrations . . . 117
7.6 fusion et composition d'activités *receive* par unification de variables . . . . 122
7.7 Exemples de composition d'activités simples . . . 123
7.8 Exemple de fusion d'invocations simples avec introduction d'un adaptateur 124
7.9 Exemple de fusion d'invocations simples en une invocation complexe . . . . 124
7.10 Exemple de compositions d'activités *reply* . . . 125

# INDEX


Action, 32, 39
  Equivalence, 47
Activité, 4, 32
  Activité composite, 32, 41
    Déterministe, 45
    Incohérente, 44
    Valide, 44
  Activité de contrôle, 109
  Activité de test, 32, 40
  Activité pivot, 56
    ↪ Interactions, 98
    ↪ Orchestrations, 135
  Activité simple, 32, 40
  Activités exclusives, 44
  Appartenance à un domaine, 45
  Diagramme d'activités en UML, 12, 33
  Dupliquée, 48
  Equivalence d'activités simples, 48
  Exécution, 32, 33, 37
ActivitéComposite
  Branche, 45
  Copie, 45
Adaptateur, 135
Aspect, 14, 119
  AspectJ, 119

Branche, 45

Composant, 7, 89
  interagissant, 114
Composition, 4
  Interprétation, 80
  De modèles, 16, 23
  De workflows, 15
  Propriétés, 24
Composition d'activités composites, 53, 65, 81
  ↪ Interactions, 91, 105
  ↪ Orchestrations, 146
  Absence de duplication, 74, 82
    ↪ Interactions, 106
    ↪ Orchestrations, 148
    Fonctions mises en jeu, 74
  Associativité, 77
    ↪ Interactions, 107
    ↪ Orchestrations, 148
    Fonctions mises en jeu♭, 77
  Clôture, 66
  Décroissance du nombre d'ensembles de fusion, 66
    ↪ Interactions, 105
    ↪ Orchestrations, 147
  Idempotence, 71, 81
    ↪ Interactions, 105
    ↪ Orchestrations, 147
    Fonctions mises en jeu, 71
  Indépendance de l'ordre, 75, 82
    ↪ Interactions, 106
    ↪ Orchestrations, 148
    Fonctions mises en jeu, 75
  Préservation des propriétés, 72, 82
    ↪ Interactions, 105
    ↪ Orchestrations, 147
    Fonctions mises en jeu, 73
  Validité, 66
Condition, 33, 43
Configuration, 111
  Application, 112

Bien construite, 111
Conflit, 54, 80, 81
Conformance, 22
Contraintes du domaine
↪ Interactions, 97
↪ Orchestrations, 135
Copie d'activité composite, 45

Domaine d'application, 38
↪ Interactions, 96
↪ Orchestrations, 131
Contraintes, 45
↪ Interactions, 97
Domaine d'interprétation, 39
Duplication, 48

Ensemble de fusion, 56
Equivalence
D'actions, 47
D'activités composites, 49
D'activités simples, 48
D'ensembles d'activités simples, 48
D'informations
↪ Interactions, 97
↪ Orchestrations, 135
De modèles, 22
De relations de condition, 48
De transformations, 59
Des copies d'activités composites, 49

Fusion d'activités simples, 61, 81
↪ Interactions, 100
↪ Orchestrations, 138
Absence d'incohérence, 64
↪ Interactions, 104
↪ Orchestrations, 145
Absence d'ordre, 63
↪ Interactions, 104
↪ Orchestrations, 145
Absence de circuit, 63
↪ Interactions, 104
↪ Orchestrations, 145
Pas de modification des activités, 62

Inclusion
D'activités composites, 49
D'ensembles d'activités simples, 48
De relations de condition, 48
De relations de précédence, 48

Information, 32, 39
↪ Interactions, 96
↪ Orchestrations, 134
Equivalence
↪ Interactions, 97
↪ Orchestrations, 135
Interaction, 89, 110, 111
*voir Configuration*, 111
*voir Règle*, 109
*voir Schéma d'interactions*, 110
Application, 112
Composition, 91, 105
Diagramme d'interactions en UML, 12
Graphe d'interactions, 90
Langage ISL, 115, 116, 118
Pose, 90, 111, 115
Schéma d'interactions, 90
Serveur d'interactions, 115
Service d'interactions, 95
Interprétation, 4, 22
Invocation complexe, 134

Métacompilation, 20

Normalisation, 49
Duplication, 50
Normalisation Partielle, 64
↪ Interactions, 104
↪ Orchestrations, 146
Normalisation sur domaine, 50
↪ Interactions, 104
↪ Orchestrations, 146

Objet, 108
Opération, 109
Orchestration, 10, 126
Composition, 126, 146
Optimisation, 127

Paire de fusion, 56
Complexité, 57
↪ Interactions, 99
↪ Orchestrations, 136
Graphes triangulés, 57
Détection, 56, 80
↪ Interactions, 98
↪ Orchestrations, 136
Indépendance de l'ordre, 57

Indépendance de l'ordre
↪ Interactions, 99
↪ Orchestrations, 136
Précédence, 33, 43
Préservation des propriétés, 72
Préservation des activités, 72
Préservation des relations, 72

Restriction des ensembles par l'équivalence, 48
Règle, 109
Instanciation, 111

Schéma d'interactions, 90, 91, 110
Application, 90
Valide, 110
Service, 126
Substitution, 47
Application, 47
Unification, 47
Système, 46

Transformation, 59, 80
↪ Interactions, 99
↪ Orchestrations, 136
Composition
↪ Interactions, 99
↪ Orchestrations, 137
Indépendance de l'ordre, 60
Composition indépendante de l'ordre
↪ Interactions, 99
↪ Orchestrations, 138
Equivalence, 59

Workflow, 9

# BIBLIOGRAPHIE


[ABV04] E. Pimentel A. Brogi, C. Canal and A. Vallecillo. Formalizing web service choreographies. In *Proc WS-FM 2004*, 2004.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava : connecting software architecture to implementation. In *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM Press.

[AD06] Samir Ammour and Philippe Desfray. A concern-based technique for architecture modelling using the uml package merge. *Electr. Notes Theor. Comput. Sci.*, 163(1) :7–18, 2006.

[Aff95] Jeff Mc Affer. Meta-level programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 190–214. Springer-Verlag, 1995.

[AL06] S. Apel and J. Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *Proceedings of ECOOP Workshop on Aspects, Dependencies, and Interactions (ADI'06)*, July 2006.

[ALB+07] Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An algebra for feature-oriented software development. Technical Report MIP-0706, University of Passau, 2007.

[All97] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. CMU Technical Report CMU-CS-97-144.

[AP07] Marcus Alanen and Ivan Porres. A metamodeling language supporting subset and union properties. *Software and Systems Modeling*, 2007.

[Bar05] Olivier Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, November 2005.

[BBF+06] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard F. Paige. A canonical scheme for model composition. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2006.

[BC89] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *OOPSLA '89 : Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM.

[BCC+06] Fabien Baligand, Hervé Chang, Pierre Combes, Laurence Duchien, Alain Ozanne, Anne-Marie Pinna-Dery, Nicolas Rivierre, Roger Rousseau, and Bruno Traverson. Chapitre Contrats : État de l'art sur la contractualisation et composition. Technical Report F.1.1, RNTL Faros, October 2006.

[BCG07] Christian Bizer, Richard Cyganiak, and Tobias Gauss. The rdf book mashup : From web apis to a web of data. In Sören Auer, Christian Bizer, Tom Heath, and Gunnar Aastrand Grimnes, editors, *SFSW*, volume 248 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[BCHS07] D. Beyer, A. Chakrabarti, T.A. Henzinger, and S.A. Seshia. An application of web-service interfaces. In *Proceedings of the International Conference on Web Services*, pages 831–838. IEEE Computer Society Press, 2007.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2003)*, Edinburgh, Scotland, May 2004.

[Ber01] Laurent Berger. *Mise en œuvre des interactions en environnements distribués, compilés et fortement typés : le Modèle MICADO*. PhD thesis, Université de Nice - Sophia Antipolis, October 2001.

[Ber03] Philip Bernstein. Applying model management to classical meta data problems. In *Conf. on Innovative Database Research (CIDR)*, Asilomar, CA, USA, jan 2003.

[Beu05] Antoine Beugnard. *Contribution à l'assemblage d'entités logicielles*. PhD thesis, Habilitation- INFO - Dépt. Informatique (Institut TELECOM/-TELECOM Bretagne), 2005.

[BF06] Mireille Blay-Fornarino. Adaptation dynamique non-anticipée des applications à base de services. In Mourad Chabane Oussalah, Flàvio Oquendo, Dalila Tamzalit, and Tahar Khammaci, editors, *1er Conférence francophone sur les Architectures Logicielles (CAL'2006)*, pages 175–176, Nantes, France, September 2006. Hermes Science.

[BFCD+06] Mireille Blay-Fornarino, Pierre Combes, Laurence Duchien, Tristan Glatard, Philippe Lahire, Stéphane Lavirotte, Clémentine Nemo, Audrey Occello, Renaud Pawlak, Anne-Marie Pinna-Déry, Lionel Seinturier, and Jean-Yves Tigli. Chapitre Développement par Composition dans Livrable Faros : État de l'art sur la contractualisation et composition. Technical Report F.1.1, RNTL Faros, October 2006.

[BFCE+04] Mireille Blay-Fornarino, Anis Charfi, David Emsellem, Anne-Marie Pinna-Déry, and Michel Riveill. Software interaction. *Journal of Object Technology (ETH Zurich)*, 3(10) :161–180, 2004.

[BFCL+07] Mireille Blay-Fornarino, Philippe Collet, Philippe Lahire, Stéphane Lavirotte, Anne-Marie Pinna-Déry, and Jean-Yves Tigli. Contrats et com-

positions de services de l'application seduite (services de diffusion ubiquitaire d'informations dans des etablissements scolaires). Technical Report F.4.1, RNTL Faros, January 2007.

[BFEO+02] Mireille Blay-Fornarino, David Emsellem, Audrey Occello, Anne-Marie Pinna-Déry, Michel Riveill, Jérémy Fierstone, Olivier Nano, and Gilles Chabert. Un service d'interactions : principes et implémentation. In *Journée composants : Systèmes à composants adaptables et extensibles*, October 2002.

[BFEPDR04] Mireille Blay-Fornarino, David Emsellem, Anne-Marie Pinna-Déry, and Michel Riveill. Un service d'interactions : principes et implémentation. *Technique et science informatiques RSTI série TSI (RSTI-TSI)*, 23(2) :175–204, 2004.

[BFFN05] Mireille Blay-Fornarino, Paul Franchi, and Olivier Nano. Vers une sémantique " plug-in " pour les modèles. In Sébastien Gérard, Jean-Marie Favre, Pierre-Alain Muller, Ivar Austvoll, and Xavier Blanc, editors, *IDM'05*, Paris, june 2005.

[BFFZ06] Mireille Blay-Fornarino and Paul Franchi-Zannettacci. *Au delà du MDA : l'Ingénierie Dirigée par les Modèles*, chapter Espace Technique "Langages" et IDM. Hermès, 2006.

[BFHJ+07] Mireille Blay-Fornarino, Vincent Hourdin, Cédric Joffroy, Stéphane Lavirotte, Sébastien Mosser, Anne-Marie Pinna-Déry, Philippe Renevier, Michel Riveill, and Jean-Yves Tigli. Architecture pour l'adaptation de Systèmes d'Information Interactifs Orientés Services. *Revues des Sciences et Technologies de l'Information (RSTI)*, pages 93–118, December 2007.

[BGB05] Salim Bouzitouna, Marie-Pierre Gervais, and Xavier Blanc. Model reuse in mda. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 354–360. CSREA Press, 2005.

[BGK+04] Michael Blow, Yaron Goland, Matthias Kloppmann, Frank Leymann, Gerhard Pfau, Dieter Roller, and Michael Rowley. Bpelj : Bpel for java. BEA and IBM, March 2004.

[Bia06] Sébastien Bianco. Vérification de l'adaptation dynamique de l'assemblage des composants. Master's thesis, DEA RSD, Nice-Sophia Antipolis, September 2006.

[BJPK+05] Alberto Bartoli, Ricardo Jiménez-Perir, Bettina Kemme, Cesar Pautasso, Simon Patarin, Stuart Wheater, and Simon Woodman. The Adapt Framework for Adaptable and Composable. *IEEE Distributed Systems Online*, 6(9), September 2005.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[BLMD06] Olivier Barais, Julia Lawall, Anne-Françoise Le Meur, and Laurence Duchien. Safe integration of new concerns in a software architecture. In *13th Annual IEEE International Conference on Engineering of Computer Based Systems ECBS'06*, Potsdam, Germany, mar 2006.

[BMO+02] M. Bartorello, Hélène Maguin, Audrey Occello, Mireille Blay-Fornarino, Anne-Marie Pinna-Déry, and Michel Riveill. Intégration de services au sein d'un serveur ejb. *RSTI - Série L'Objet (RSTI-Objet)*, 8(1-2) :169–184, January 2002.

[Boc03] Conrad Bock. Uml 2 activity and action models. *Journal of Object Technology*, 2(4) :43–53, 2003.

[BP01] Dusan Bálek and Frantisek Plasil. Software connectors and their role in component deployment. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 69–84, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[Bri89] Jean-Pierre Briot. Actalk : A testbed for classifying and designing actor languages in the smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP'89*, pages 109–129, Nottingham, 1989. Cambridge University Press.

[BSL01] Noury M. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. *Technique et Science Informatique*, 20(4), 2001.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[Bus00] Laurent Bussard. Towards a pragmatic composition model of Corba services based on AspectJ. Master's thesis, Nice University, Nice-Sophia Antipolis, 2000.

[BW03] Wolf-Tilo Balke and Matthias Wagner. Cooperative discovery for user-centered web service provisioning. In Zhang [Zha03], pages 191–197.

[Car03] Eric Cariou. *Contribution à un Processus de Réification d'Abstractions de Communication*. PhD thesis, Université de Rennes, June 2003.

[Cas05] Yves Caseau. Self-adaptive middleware : Supporting business process priorities and service level agreements. *Advanced Engineering Informatics*, 19(3) :199–211, 2005.

[CCCV05] Javier Camara, Carlos Canal, Javier Cubo, and Antonio Vallecillo. Formalizing WSBPEL Business Processes Using Process Algebra. In *Foundations of Coordination Languages and Software Architectures (FOCLASA)*, San Francisco (CA), August 2005. Springer.

[CCMN04] Girish B. Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *WWW Alt. '04 : Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.

[CF05a] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In *ICSE '05 : Proceedings of the 27th international conference on Software engineering*, pages 69–77, New York, NY, USA, 2005. ACM Press.

[CF05b] Carine Courbis and Anthony Finkelstein. Weaving aspects into web service orchestrations. In *ICWS*, pages 219–226. IEEE Computer Society, 2005.

[CFWBFT+05] Daniel Cheung Foo Woo, Mireille Blay-Fornarino, Jean-Yves Tigli, Anne-Marie Pinna-Déry, David Emsellem, and Michel Riveill. Langage d'aspects pour la composition dynamique de composants embarqués. In Pierre Cointe and Lionel Seinturier, editors, *JFDLPA'05*, Lille, France, sep 2005.

[Cla02] Siobhàn Clarke. Extending standard uml with model composition semantics. *Sci. Comput. Program.*, 44(1) :71–100, 2002.

[CM04] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In *ECOWS*, volume 3250 of *LNCS*, pages 168–182. Springer, 2004.

[CM05] Anis Charfi and Mira Mezini. An aspect-based process container for bpel. In *AOMD '05 : Proceedings of the 1st workshop on Aspect oriented middleware development*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[COR06] Philippe Collet, Alain Ozanne, and Nicolas Rivierre. On contracting different behavioral properties in component-based systems. In *ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27*, 2006.

[CP05] Praveen Chandran and Arun Poduval. Adding BPEL to the Enterprise Integration Mix. Technical report, ORACLE, November 2005.

[CRBFPD06] Anis Charfi, Michel Riveill, Mireille Blay-Fornarino, and Anne-Marie Pinna-Déry. Transparent and Dynamic Aspect Composition. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, Bonn, Germany, March 2006.

[CRCR05] P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre. A Contracting System for Hierarchical Components. In *ICSE'2005 - CBSE8*. Springer - LNCS, 2005.

[Dah01] M. Dahm. Byte code engineering with the bcel api, 2001.

[DBFA+01] Anne-Marie Dery, Mireille Blay-Fornarino, Borice Arcier, Léonard Mule, and Sabine Moisan. Distributed access knowledge-based system : Reified interaction service for trace and control. In *DOA*, pages 76–84, 2001.

[DBFP95] Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna. A reflective model for first class dependencies. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 265–280. ACM, oct 1995.

[Des08] Nicolas Desnos. *Ports composites pour l.* PhD thesis, Université Montpellier II, Nîmes (France), June 2008.

[dF07] Grégory de Fombelle. *Gestion incrémentale des propriétés de cohérence structurelle dans l'ingénierie dirigée par les modèles*. PhD thesis, université Pierre et Marie Curie, Paris VI, September 2007.

[DFL+05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *AOSD'05 : Proceedings*

*of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[DFQJ08] Bruno De Fraine, Pablo Daniel Quiroga, and Viviane Jonkers. Management of aspect interactions using statically-verified control-flow relations. In *Proceedings of 3rd Int. Workshop on Aspects, Dependencies and Interactions, ADI'08*, Paphos, Cyprus, July 2008. AOSD.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04 : Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[Duc97] Stéphane Ducasse. *Intégration de relations comportementales dans un modèle objet à classes*. PhD thesis, Université de Nice - Sophia Antipolis, January 1997.

[Duc02] Roland Ducournau. Spécialisation et sous-typage : thème et variations. *Technique et science informatiques RSTI série TSI (RSTI-TSI)*, 21(10) :1305–1342, oct 2002.

[DUVH06] Nicolas Desnos, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. Assistance à l'architecte pour la construction d'architectures à base de composants. In Rousseau et al. [RUV06], pages 37–52.

[DÜYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems Inc., version 2.0 edition, August 2001.

[EBR00] A. Cavarra E. Borger and E. Riccobene. An asm semantics for uml activity diagrams. In *Proc Algebraic Methodology and Software Technology (AMAST 2000), LNCS 1816*, 2000.

[EW04] R. Eshuis and R. Wieringa. Tool support for verifying uml activity diagrams. In *IEEE Transactions on Software Engineering, vol 30*, 2004.

[FEBF06] Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles : au-delà du MDA*. Hermes-Lavoisier, Cachan, France, February 2006.

[FF05] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *AOSD'05*, pages 21–35. Addison-Wesley, Boston, 2005.

[FFR+07] Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, Annapolis, MD, USA, 2007.

[FSJ08] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ : flexible and safe pointcut/advice bindings. In Theo D'Hondt, editor, *AOSD*, pages 60–71. ACM, 2008.

[FTS06] Régis Fleurquin, Chouki Tibermacine, and Salah Sadou. Une assistance pour l'évolution des logiciels à base de composants. *l'Objet*, 2006.

[FUMK03] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pages 152–163. IEEE Computer Society, 2003.

[FUMK04] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility verification for web service choreography. In *ICWS '04 : Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 738, Washington, DC, USA, 2004. IEEE Computer Society.

[FV07] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-automatic model integration using matching transformations and weaving models. In *SAC '07 : Proceedings of the 2007 ACM symposium on Applied computing*, pages 963–970, New York, NY, USA, 2007. ACM.

[FZ82] Paul Franchi-Zannettacci. *Attributs sémantiques et schémas de programmes*. Thèse d'état, University de Bordeaux I, March 1982.

[GBG05] Walid Gaaloul, Karim Baïna, and Claude Godart. Towards mining structural workflow patterns. In Kim Viborg Andersen, John K. Debenham, and Roland Wagner, editors, *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 24–33. Springer, 2005.

[GEM06] Tristan Glatard, David Emsellem, and Johan Montagnat. Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. In *Grid-Enabling Legacy Applications and Supporting End Users Workshop (GELA'06)*, Paris, France, June 2006.

[GMP+06] Tristan Glatard, Johan Montagnat, Xavier Pennec, David Emsellem, and Diane Lingrand. MOTEUR : a data-intensive service-based workflow manager. Technical Report I3S/RR-2006-07-FR, I3S, Sophia Antipolis (France), March 2006.

[Ham08] Douae Hammouche. Model Driven orchestration Composition. Master's thesis, Mohammed V, Agdal University Rabat Morocco, Nice-Sophia Antipolis , Rabat, July 2008.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An algebraic view on the semantics of model composition. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2007.

[JEA+07] Diane Jordan, John Evedmon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Fransisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Kevin Liu, Rania Khalaf, Dieter Konig, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny Van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. Technical report, OASIS, 2007.

[JFBF06] Estublier Jacky, Jean Marie Favre, and Mireille Blay-Fornarino, editors. *Au delà du MDA : l'Ingénierie Dirigée par les Modèles* . Hermès, 2006.

[JMBF07] Cédric Joffroy, Sébastien Mosser, and Mireille Blay-Fornarino. Plateforme ADORE : Aspect and Distributed ORchEstrations. Technical report, I3S, Sophia-Antipolis (France), March 2007.

[JMBFN07] Cédric Joffroy, Sébastien Mosser, Mireille Blay-Fornarino, and Clémentine Nemo. Des Orchestrations de Services Web aux Aspects. In *3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFLDPA'2007)*, Toulouse (France), March 2007.

[KFJ07] Jacques Klein, Franck Fleurey, and Jean Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *Transactions on Aspect-Oriented Software Development (TAOSD)*, LNCS 4620 :167–199, 2007.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, June 2001.

[KHJ06] Jacques Klein, Loic Hélouet, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, March 2006. ACM.

[Kli08] Paul Klint. Tribute to a great meta-technologist : from centaur to the meta-environment. In *From semantics to computer science : essays in honor of Gilles Kahn*. Cambridge University Press, June 2008.

[KM05] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[KMW03] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-oriented composition in bpel4ws. In *WWW (Alternate Paper Tracks)*, 2003.

[LCL06] Marc Léger, Thierry Coupaye, and Thomas Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In Rousseau et al. [RUV06], pages 21–36.

[LHB05] Roberto E. Lopez-Herrejon and Don Batory. Taming aspect composition : A functional approach. Technical Report UTEXAS.CS//CS-TR-05-27, Department of Computer Sciences The University of Texas at Austin, 1 University Station, C0500 Austin, TX 78712, June 2005.

[LHBL06] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06 : Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM Press.

[LV95] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pages 717–734, September 1995.

[Mar05] Axel Martens. Analyzing Web Service based Business Processes. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442, Edinburgh (Scotland), April 2005. Springer-Verlag.

[MB05] Selma Matougui and Antoine Beugnard. How to implement software connectors ? a reusable, abstract and adaptable connector. In Lea Kutvonen and Nancy Alonistioti, editors, *DAIS*, volume 3543 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2005.

[MBFR08] Sébastien Mosser, Mireille Blay-Fornarino, and Michel Riveill. Web Services Orchestration Evolution : A Merge Process For Behavioral Evolution. In *2nd European Conference on Software Architecture (ECSA'08)*, Paphos, Cyprus, September 2008. Springer LNCS.

[MCBFR08] Sébastien Mosser, Franck Chauvel, Mireille Blay-Fornarino, and Michel Riveill. Web Service Composition : Mashups Driven Orchestration Definition. In *International Conference on Itelligent Agents, Web Technologies and Internet Commerce (IAWTIC'08)*, Vienna, Austria, December 2008. IEEE Computer Society.

[McG82] James R. McGraw. The val language : Description and analysis. *ACM Trans. Program. Lang. Syst.*, 4(1) :44–82, 1982.

[MD03] Philippe Mougin and Stéphane Ducasse. OOPAL : Integrating array programming in object-oriented programming. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 65–77, oct 2003.

[MDBF06] Raphaël Marvie, Laurence Duchien, and Mireille Blay-Fornarino. *Au delà du MDA : l'Ingénierie Dirigée par les Modèles*, chapter Les plates-formes d'exécution et l'IDM. Hermès, 2006.

[MGL06] Johan Montagnat, Tristan Glatard, and Diane Lingrand. Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France, June 2006.

[MHS06] Eetu Mäkelä, Eero Hyvönen, and Samppa Saarela. Ontogator - a semantic view-based search engine service for web applications. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 847–860. Springer, 2006.

[MKR06] Tom Mens, Günter Kniesel, and Olga Runge. Transformation dependency analysis. A comparison of two approaches. In *Langages et Modèles à Objets (LMO)*, pages 167–182, Nimes, March 2006. Hermes.

[MLM+06] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, February 2006.

[Nan01] Olivier Nano. Composition de services. Master's thesis, Université de Nice-Sophia Antipolis, Sophia Antipolis, June 2001.

[Nan04] Olivier Nano. *Un modèle de réécriture pour l'intégration de services*. PhD thesis, Université de Nice - Sophia Antipolis, Sophia Antipolis, France, 2004.

[NBF03a] O. Nano and M. Blay-Fornarino. Using mda to integrate services in component platforms. In *Eighth International Workshop on Component-Oriented Programming (WCOP 2003) in conjuction with ECOOP2003*, Darmstat, Germany, 21 july 2003.

[NBF03b] Olivier Nano and Mireille Blay-Fornarino. Services integration by model annotation and transformation. In *First International Workshop on Metamodelling for MDA*, pages 77–92, York, England, November 2003. A. Evans, P. Sammut and J. Willan.

[NBF04] Olivier Nano and Mireille Blay-Fornarino. Annotations et transformations de modèles pour l'intégration de services. *RSTI - Série L'Objet (RSTI-Objet)*, 10(2-3) :175–188, 2004.

[NBFKR07] Clémentine Nemo, Mireille Blay-Fornarino, Günter Kniesel, and Michel Riveill. SEMANTIC ORCHESTRATIONS MERGING - Towards Composition of Overlapping Orchestrations. In Joaquim Filipe, editor, *9th International Conference on Enterprise Information Systems (ICEIS'2007)*, Funchal, Madeira, June 2007.

[NBFR08] Clémentine Nemo, Mireille Blay-Fornarino, and Michel Riveill. Multi-Applications de Patterns - Formalisation. In *4ième Journée sur l'Ingénierie Dirigée par les Modèles (IDM'08)*, Mulhouse (France), June 2008.

[Nem06] Clémentine Nemo. Vers la composition d'orchestrations de services. Master dissertation, DEA PLMT, Nice, France, June 2006.

[NGBFM07] Clémentine Nemo, Tristan Glatard, Mireille Blay-Fornarino, and Johan Montagnat. Merging overlapping orchestrations : an application to the Bronze Standard medical application. In *International Conference on Services Computing (SCC 2007)*, Salt Lake City, Utah, USA, July 2007. IEEE Computer Engineering.

[NSC+07] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *ICSE '07 : Proceedings of the 29th international conference on Software Engineering*, pages 54–64, Washington, DC, USA, 2007. IEEE Computer Society.

[OAF+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna : A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20) :3045–3054, 2004.

[obj05] objectweb. The OMG's CORBA Components Technology. http ://openccm.objectweb.org/doc/ccm.html, 2005.

[Obj08] Consortium ObjectWeb. JOnAS : JavaTM Open Application Server, July 2008. http ://www.objectweb.org/jonas.

[Occ02] Audrey Occello. Composants : Vers une adaptation dynamique cohérente. Master's thesis, DEA Informatique, Nice-Sophia Antipolis, July 2002.

[Occ06] Audrey Occello. *Capitalisation de la sûreté de fonctionnement des applications soumises aux adaptations dynamiques : le modèle exécutable Satin*. PhD thesis, Universite de Nice - Sophia Antipolis, Sophia Antipolis (France), June 2006.

[OMG07] OMG. Unified modeling language : Superstructure (version 2.1.1). http ://www.omg.org/docs/formal/07-02-03.pdf, february 2007.

[OPD04] Audrey Occello and Anne-Marie Pinna-Déry. An Adaptation-safe Model for Component Platforms. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE)*, Nice (France), jul 2004.

[Pas06a] Robert Pastor. Model Composition : Definition of Model Composition Properties. Technical Report D1.5, IST, University of York, March 2006.

[Pas06b] Robert Pastor. Model Composition : Model Transformation for Model Composition. Technical Report D1.5-3, University of York, September 2006.

[PB05] C. Prehofer and C. Bettstetter. Self-organization in communication networks : principles and design paradigms. *Communications Magazine, IEEE*, 43(7) :78–85, 2005.

[PDF03] Anne-Marie Pinna-Déry and Jérémy Fierstone. Component model and programming : a first step to manage Human Computer Interaction Adaptation. In *5th International Symposium on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)*, volume LNCS 2795, pages 456–460, Udine, Italy, September 2003. L. Chittaro (Ed.), Springer Verlag.

[PDS05] Renaud Pawlak, Laurence Duchien, and Lionel Seinturier. Compar : Ensuring safe around advice composition. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005, Proceedings*, volume 3535 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2005.

[Pel03] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10) :46–52, 2003.

[Pes07] Nicolas Pessemier. *Unification des approches par aspects et à composants*. PhD thesis, Université Lille 1, Laboratoire d'Informatique Fondamentale de Lille, June 2007.

[Pla05] Frantisek Plasil. Enhancing component specification by behavior description : the sofa experience. In *WISICT '05 : Proceedings of the 4th international symposium on Information and communication technologies*, pages 185–190. Trinity College Dublin, 2005.

[Pou07] Frédéric Pourraz. *Diapason : une approche formelle et centrée architecture pour la composition évolutive de services Web*. PhD thesis, Université de Savoie, LISTIC, december 2007.

[PS05] Victor Pankratius and Wolffried Stucky. A formal foundation for workflow composition, workflow view definition, and workflow normalization based on petri nets. In *APCCM '05 : Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*, pages 79–88, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[PSDB04] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Olivier Barais. Une extension de fractal pour l'aop. In *Première journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA 2004)*, Paris, France, September 2004.

[PSDC06] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, Lecture Notes in Computer Science. Springer, March 2006.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and G. Florin. Jac : A flexible and efficient solution for aspect-oriented programming in java. In *Reflection 2001*, volume 2194 of *Lecture Notes in Computer Sciences*, page 1Ű24, Kyoto, Japan, September 2001. Springer Verlag.

[PSSN03] Massimo Paolucci, Naveen Srinivasan, Katia P. Sycara, and Takuya Nishimura. Towards a semantic choreography of web services : From wsdl to daml-s. In Zhang [Zha03], pages 22–26.

[PTBM05] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated synthesis of composite bpel4ws web services. In *ICWS*, pages 293–301. IEEE Computer Society, 2005.

[Rap02] Pascal Rapicault. *Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation*. PhD thesis, Université de Nice - Sophia Antipolis, May 2002.

[RBY+04] R. Razavi, N. Bouraqadi, J. W. Yoder, J. F. Perrot, and R. Johnson. Language support for adaptive object-models using metaclasses. In *Research Track of the ESUG 2004 Smalltalk Conference*, Köthen (Anhalt), Germany, September 2004. Selected for publication in the Elsevier international journal "Computer Languages, Systems and Structures".

[RDHS06] Jinghai Rao, Dimitar Dimitrov, Paul Hofmann, and Norman Sadeh. A mixed initiative approach to semantic web service discovery and composition : Sap's guided procedures framework. In *ICWS '06 : Proceedings of the IEEE International Conference on Web Services*, pages 401–410, Washington, DC, USA, 2006. IEEE Computer Society.

[RPPM06] Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, and Philippe Merle. Using attribute-oriented programming to leverage fractal-based developments. In *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal'06)*, Nantes, France, jul 2006. To appear.

[RRB02] P. Rapicault, J-P. Rigault, and L. Bourlier. Model, notation, and tools for verification of protocol-based components assembly. In Judith Bishop, editor, *Component Deployment* $CD2002$, number 2370 in LNCS, pages 257–268. IFIP/ACM Working Conference, Springer-Verlag, June 2002.

[Rub08] Daniel Rubio. BPEL : Web Services orchestration, hands-on with ActiveBPEL, 2008. http ://www.webforefront.com/about/danielrubio/articles/techtarget/bpel_activebpel.html

[RUV06] Roger Rousseau, Christelle Urtado, and Sylvain Vauttier, editors. *Langages et Modèles à objets*, Nîmes (France), March 2006. Hermès - Lavoisier.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4) :314–335, 1995.

[SISM02] BEA Systems, Intalio, SAP, and Sun Microsystems. Web service choreography interface (wsci) 1.0. http ://www.w3.org/TR/wsci/, August 2002.

[SJ05] Jim Steel and Jean-Marc Jézéquel. Model typing for improving reuse in model-driven engineering. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 84–96. Springer, 2005.

[SM02] Clemens Szyperski and Stephan Murer. *Component software : beyond object-oriented programming 2nd ed.* Addison-Wesley, 2002.

[Sti81] Mark E. Stickel. A unification algorithm for associative-commutative functions. *J. ACM*, 28(3) :423–434, 1981.

[Szy96] Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

[TBB04] Francis Tessier, Mourad Badri, and Linda Badri. A model-based detection of conflicts between crosscutting concerns : Towards a formal approach. In Minhuan Huang, Hong Mei, and Jianjun Zhao, editors, *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, September 2004.

[TDG+07] Frédéric Thomas, Jérôme Delatour, Sébastien Gérard, Matthias Brun, and François Terrier. Contribution à la modélisation explicite des plates-formes d'exécution pour l'IDM. *L'objet*, 13 :9–32, April 2007.

[TFS04] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Préservation de choix architecturaux lors de l'évolution d'un composant. In *Proceedings of OCM-SI'04 workshop (Objets, Composants et Modèles pour les Systèmes d'Information), held in conjunction with INFORSID'04*, Biaritz, France, May 2004.

[TTT06] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1) :28–42, 2006.

[vdAvDH+03] W.M.P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining : a survey of issues and approaches. *Data and Knowledge Engineering*, 47(2) :237–267, 2003.

[VDS05] Ragnhild Van Der Straeten. *Inconsistency management in model-driven engineering : an approach using description logics*. PhD thesis, Vrije Universiteit Brussel, 2005.

[Vir04] Mirko Viroli. Towards a formal foundation to orchestration languages. *Electr. Notes Theor. Comput. Sci.*, 105 :51–71, 2004.

[VVJ06] Bart Verheecke, Wim Vanderperren, and Viviane Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1) :42–50, 2006.

[Wag08] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2008.

[Zha03] Liang-Jie Zhang, editor. *Proceedings of the International Conference on Web Services, ICWS '03, June 23 - 26, 2003, Las Vegas, Nevada, USA*. CSREA Press, 2003.

Interprétations de la composition d'activités

**Résumé**

Nos travaux de recherche portent sur la composition fiable d'activités logicielles. Ils s'inscrivent dans le cadre général du génie logiciel, et concernent les étapes de conception, implantation et adaptations d'applications logicielles réparties. Le thème central développé est la modélisation des activités et leur composition tout en garantissant différentes propriétés. La spécificité de ce travail et son originalité est d'avoir suivi une approche de la composition non pas dirigée par les langages mais par les éléments clefs de la composition indépendamment de mises en œuvres spécifiques. En fondant ce travail sur une approche formelle, nous proposons une vision unifiée d'un procédé de composition, et caractérisons les différences en terme d'interprétations. En étayant cette formalisation par plusieurs mises en œuvre à la fois en terme d'environnement de programmation et d'applications nous ancrons cette approche dans une réalité fonctionnelle.
Les principales applications de ce travail portent sur la composition d'interactions entre composants hétérogènes et la composition de workflows.