Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : application aux services de transactions

Romain Rouvoy

▶ To cite this version:

Romain Rouvoy. Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : application aux services de transactions. domain_stic.inge. Université des Sciences et Technologie de Lille - Lille I, 2006. Français. NNT : . tel-00119794

HAL Id: tel-00119794
https://theses.hal.science/tel-00119794v1

Submitted on 12 Dec 2006

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.

Département de formation doctorale en informatique
UFR IEEA

École Doctorale SPI Lille

Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : *application aux services de transactions*

THÈSE

présentée et soutenue publiquement le 08 décembre 2006

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Romain ROUVOY

Composition du jury :

Président du jury :	M. Nouredine MELAB, Professeur	LIFL / Université de Lille I
Rapporteurs :	M. Jean-Marc JÉZÉQUEL, Professeur	IRISA / Université de Rennes I
	M. Jean-Bernard STEFANI, Directeur de recherche	INRIA Rhône-Alpes
Examinateurs :	M. Thomas LEDOUX, Enseignant-Chercheur	École des Mines de Nantes
	M. Bruno TRAVERSON, Chercheur	EDF Recherche & Développement
Directeurs de thèse :	M. Jean-Marc GEIB, Professeur	LIFL / Université de Lille I
	M. Philippe MERLE, Chargé de recherche	LIFL / INRIA FUTURS

INRIA FUTURS - Laboratoire d'Informatique Fondamentale de Lille - UMR USTL/CNRS 8022

Résumé

Cette thèse adresse la problématique de la construction des intergiciels hautement adaptables. Ces intergiciels se caractérisent par une grande diversité des fonctionnalités fournies. Dans le domaine des transactions, cette diversité concerne non seulement les modèles de transactions, les protocoles de contrôle de concurrence et de reprise après défaillance, mais aussi les normes et les standards d'intégration. Notre proposition consiste à définir un canevas intergiciel capitalisant la diversité du domaine transactionnel, et permettant de construire des services de transactions hautement adaptables. Ce type de services requiert la mise en place d'une démarche de construction à granularité extrêmement fine afin de pouvoir adapter les nombreuses caractéristiques de l'intergiciel.

Nous proposons donc de compléter l'approche issue des *exogiciels* avec quatre nouveaux éléments. Ainsi, nous définissons le modèle de programmation FRACLET à base d'annotations pour favoriser la programmation des abstractions fonctionnelles de l'intergiciel. Nous proposons ensuite un langage de description et de vérification de motifs d'architecture pour fiabiliser la modélisation des abstractions architecturales. Ces deux premiers éléments servent à la conception d'un canevas intergiciel à base de composants utilisant les motifs de conception comme structure architecturale extensible. Enfin, nous décrivons les configurations possibles en utilisant différents modèles de haut niveau dédiés aux caractéristiques de l'intergiciel. Nous illustrons ces concepts en présentant GOTM, un canevas intergiciel à composants pour la construction de services de transactions hautement adaptables.

Notre approche est validée au travers de trois expériences originales. Tout d'abord, nous proposons de faciliter l'intégration des services de transactions dans les plates-formes intergicielles par la définition de politiques de démarcation transactionnelle indépendantes de la plate-forme et du type de service intégré. Ensuite, nous définissons un service de transactions composant plusieurs personnalités simultanément pour faciliter l'interopérabilité transactionnelle d'applications hétérogènes. Enfin, nous sommes en mesure de sélectionner différents protocoles de validation à deux phases pour optimiser le temps d'exécution des transactions face aux changements des conditions d'exécution de l'application.

Abstract

This thesis focuses on the construction of highly adaptable middleware services. These services are characterized by a large diversity of the provided functions. In the transaction domain, this diversity includes the transaction models, the concurrency control and fault tolerance protocols, and the integration standards. Our proposition consists in defining a middleware framework that covers the transaction domain diversity, and allows the construction of highly adaptable transaction services. This kind of services requires to define an extremely fined-grained approach in order to adapt their various characteristics.

Thus, we propose to complete the *exoware* approach with four new contributions. We define the FRACLET programming model to leverage the programming of middleware functional abstractions. Then, we propose a language to describe and verify architectural patterns in order to design the middleware architectural abstractions. These two contributions allow us to define a component-based middleware framework that uses design patterns to build a modular and extensible architectural substrate. Finally, we configure our middleware framework using various high level models dedicated to each middleware concern. We illustrate these concepts on GOTM, a component-based middleware framework for the construction of highly adaptable transaction services.

We validate our approach on three original experiences. Firstly, we propose to leverage the integration of transaction services into the middleware platforms by designing demarcation policies independently of the platform and the kind of services that are integrated. Secondly, we define a transaction service that composes several personalities to seamlessly support the transaction interoperability between heterogeneous applications. Thirdly, we are able to select the best performing two-phase commit protocol at runtime to improve the transaction completion time depending on the evolution of the execution context.

Remerciements

Je tiens à remercier en premier lieu les membres de mon jury. En particulier, je remercie Nouredine Melab pour m'avoir fait le plaisir de présider ce jury. J'adresse ensuite un grand merci à Jean-Marc Jézéquel et à Jean-Bernard Stefani pour avoir accepté de rapporter mes travaux. La qualité de ces rapports et la pertinence des remarques m'ont permis d'améliorer la qualité de mon manuscrit. Enfin, je tiens à remercier chaleureusement Thomas Ledoux et Bruno Traverson d'avoir accepté d'examiner mon travail.

Mes remerciements vont ensuite à Jean-Marc Geib, directeur du laboratoire et responsable de l'équipe GOAL et du projet INRIA JACQUARD, mais surtout encadrant de ma thèse. Malgré un emploi du temps très chargé, tu as su me consacrer du temps aux moments importants.

Un grand merci à Philippe Merle, dont l'encadrement durant toutes ces années a été sans faille. Au delà de nos nombreuses discussions, tu m'as transmis ta passion pour la recherche au point de me convaincre d'entreprendre cette thèse. De plus, tu as toujours pris le temps de me conseiller au mieux et de m'encourager lors de mes périodes de doute. Sans ton soutien continuel, cette thèse ne serait pas ce qu'elle est.

Je tiens à adresser une mention spéciale à Denis Conan, inconditionnel de Fraclet[1]. Lors de ton passage à Lille, ta rigueur et ton organisation m'ont permis d'améliorer notablement la qualité de mes travaux. Je te remercie encore pour la relecture méticuleuse de mon manuscrit et tes remarques très judicieuses. J'espère que nos collaborations seront à l'avenir encore très nombreuses.

Je remercie ensuite tous les membres du projet JACQUARD et de l'équipe GOAL.

- En particulier, je remercie Laurence Duchien, Anne-Françoise Le Meur, et Lionel Seinturier pour nos nombreuses et fructueuses discussions. Merci également à Raphaël Marvie, qui m'a montré le chemin du bureau 221 et avec qui j'ai pu partager mon expérience d'enseignant pendant trois années. Mais ces remerciements s'adressent également à Olivier Caron, Bernard Carré, Nicolas Dolet, Maja D'Hondt, Johan Fabry, Areski Flissi, Jean-François Roos, Missi Tran-Ahn, et Gilles Vanwormhout.
- De plus, je souhaite beaucoup de réussite à mes collègues apprentis chercheurs : Dolorès Diaz, Nicolas Pessemier, Frédéric Loiret, Naouel Moha, Jérémy Dubus, Carlos Noguera, Aleš Plšek, Guillaume Dufrêne, et Guillaume Waignier. Il est normal de douter mais l'issue en vaut la peine !
- Je remercie Patricia Serrano-Alvarado dont le passage dans l'équipe a permis de dynamiser les travaux autour de la thématique des transactions. Merci pour ta bonne humeur et ton amitié.

[1] Fraclet est une des contributions de ma thèse présentée dans le chapitre 5.

– Je ne peux oublier les anciens de l'équipe que j'ai eu le plaisir de cotoyer durant ces années : Olivier Barais, Johan Brichau, Éric Cariou, Christophe Contreras, Christophe Demarey, Bassem Kosayba, Julia Lawall, Sylvain Leblanc[2], Alexis Muller, Nicolas Petiprez, Emmanuel Renaux, Renaud Pawlak, et Mathieu Vadet.
– Je remercie tous les membres[3] du bureau 326 présents et passés pour nos nombreuses discussions et leur aide précieuse lors de ma rédaction. Nos relations ont dépassé à mon sens les simples relations de collègues de bureau et j'espère que cette amitié perdurera.
– Je remercie les membres du bureau de l'association TILDA[4] pour l'aide précieuse qu'ils m'ont apporté dans la mise en place de l'association. J'espère que cette association tiendra son objectif de rapprocher les doctorants du laboratoire.
– J'associe à ces remerciements Corinne Davoust, Karine Lewandowski et Axelle Magnier, nos assistantes de projet, et Marie-Agnès Enard pour leur gentilesse, leur patience et leur aide de tous les instants.
– De façon plus générale, je remercie tous les membres du laboratoire d'informatique fondamentale de Lille pour leur accueil et l'aide qu'ils m'ont apportés à différents instants de mon passage au LIFL.

Sur le plan non scientifique, je tiens à remercier mes amis de toujours Pierre Carpentier et Jérôme Moroy sans oublier leurs compagnes respectives Faustine et Élodie. C'est un grand bonheur de partager votre amitié depuis toutes ces années, et pourvu que ça dure !

Je remercie mes amis parisiens Benoît Delplancke et Pascal Ogil, ainsi que mes compagnons de DEA Alexandre Courbot et Joël Vennin avec lesquels je conserve des liens depuis la première année de DUT, soit 8 ans maintenant.

Je remercie Nathalie et Franck Lecoq non seulement pour l'intérêt qu'ils ont su porter à mes études mais aussi pour l'accueil chaleureux qu'ils m'ont réservé au sein de leur foyer.

Je remercie également les membres de ma famille pour leur soutien constant tout au long de mes études. En particulier, je ne saurais comment remercier mes parents et mon petit-frère Rémy pour les encouragements et le soutien qu'ils n'ont cessés de m'apporter depuis toujours. Vous m'avez toujours encouragé à poursuivre mes études et, grâce à vous, cette thèse est aussi la vôtre.

Enfin, un grand merci à Élodie, mon petit ange, pour avoir supporté mon mauvais caractère et m'avoir offert une oreille attentative dans mes périodes de doute. Ta présence et ton amour au quotidien sont pour moi un précieux réconfort.

À tous, un grand merci !

[2]J'espère que ces remerciements te parviendront car on ne t'oublie pas.
[3]Nul besoin de les nommer de nouveau car ils se reconnaîtront.
[4]Association des Thésards en Informatique de Lille et Docteurs Associés.

Table des matières

1 Introduction **1**
1.1 Contexte du travail . . . 1
1.2 Problématique . . . 2
1.2.1 Axes d'évolution des services de transactions . . . 2
1.2.2 Problématique de la construction d'intergiciels hautement adaptables . . . 5
1.3 Proposition . . . 6
1.3.1 Granularité de l'adaptation . . . 6
1.3.2 Contributions à la conception des services de transactions . . . 8
1.3.3 Validation des services de transactions construits avec GOTM . . . 9
1.4 Organisation du document . . . 9
1.4.1 État de l'art . . . 9
1.4.2 Contributions . . . 9
1.4.3 Validation . . . 10

I État de l'art **11**

2 Adaptabilité dans les intergiciels transactionnels **13**
2.1 Introduction . . . 14
2.2 Prérequis . . . 14
2.2.1 Définition d'une transaction . . . 14
2.2.2 Propriétés ACID . . . 14
2.2.3 Modèles de transactions . . . 15
2.2.4 Atomicité globale . . . 17
2.2.5 Services de transactions . . . 20
2.2.6 Standards transactionnels . . . 21
2.2.7 Synthèse . . . 23
2.3 Étude de l'existant . . . 23
2.3.1 Critères d'évaluation . . . 24
2.3.2 *Reflective Transaction Framework* . . . 25
2.3.3 *Kernel Aspect Language for ATMS* . . . 26
2.3.4 REFLECTS . . . 28
2.3.5 Service d'adaptation des services techniques . . . 30
2.3.6 *Arjuna Transaction System* . . . 31
2.3.7 Autres approches . . . 32
2.4 Synthèse . . . 33
2.5 Conclusion . . . 36

3 Modèles de composants pour intergiciels 37
3.1 Introduction . . . 37
3.2 Prérequis . . . 38
3.2.1 Concepts fondamentaux . . . 38
3.2.2 Conception à base de composants . . . 39
3.2.3 Architectures logicielles . . . 40
3.2.4 Canevas intergiciels . . . 40
3.3 Étude de l'existant . . . 41
3.3.1 Modèle de composants FRACTAL . . . 41
3.3.2 Modèle de composants OPENCOM . . . 46
3.3.3 Modèle de composants ABC . . . 49
3.3.4 Autres approches . . . 53
3.4 Conclusion . . . 56

II Contributions 59

4 Démarche de construction de GOTM 61
4.1 Introduction . . . 62
4.2 Motivations . . . 62
4.2.1 Limitations des canevas logiciels transactionnels actuels . . . 63
4.2.2 Objectifs d'un canevas de services de transaction hautement adaptables . . . 63
4.2.3 Démarches de construction envisageables . . . 64
4.2.4 Approche de construction à très fine granularité . . . 65
4.3 Vers des composants de très fine granularité . . . 66
4.3.1 Principes de conception des composants de très fine granularité . . . 66
4.3.2 Illustration de notre démarche de construction . . . 67
4.4 Présentation du canevas logiciel GOTM . . . 70
4.4.1 Démarche de construction de GOTM . . . 70
4.4.2 Modèle de programmation des composants . . . 70
4.4.3 Langage de description et de vérification de motifs d'architecture . . . 71
4.4.4 Bibliothèque de composants transactionnels GOTM . . . 72
4.4.5 Modèles dédiés à la configuration des services de transactions . . . 72
4.5 Organisation de la deuxième partie . . . 72

5 Modèle de programmation de composants 73
5.1 Motivations . . . 74
5.1.1 Entrelacement du code métier et technique . . . 74
5.1.2 Redondance des méta-informations . . . 76
5.2 Principes de la programmation par attributs . . . 77
5.3 Application au modèle de composants FRACTAL . . . 78
5.3.1 Description des annotations FRACLET . . . 78
5.3.2 Description des générateurs FRACLET . . . 80
5.3.3 Application HelloWorld revisitée . . . 81
5.4 Éléments d'implantation de FRACLET . . . 83
5.4.1 Description du moteur de génération . . . 83
5.4.2 Illustration du code généré . . . 84
5.5 Évaluations . . . 87
5.5.1 Évaluation quantitative du code source . . . 87
5.5.2 Évaluation quantitative du code généré . . . 88
5.5.3 Évaluation qualitative . . . 89
5.6 Généralisation à d'autres modèles de composants . . . 90
5.6.1 Programmation orientée composant et dirigée par le code . . . 90
5.6.2 Description des générateurs OPENCOM . . . 91

5.6.3 Illustration du code généré pour OPENCOM 91
5.6.4 Évaluation 91
5.7 Travaux connexes 94
5.7.1 Programmation par attributs 94
5.7.2 Programmation générative 94
5.7.3 Programmation par aspects 95
5.7.4 Langages dédiés 95
5.8 Conclusion 96

6 Description et vérification de motifs d'architecture 99
6.1 Introduction 100
6.2 Motivations 101
6.2.1 Prise en compte des constructions récurrentes 101
6.2.2 Séparation des préoccupations 102
6.3 Langage de description et de vérification de motifs d'architecture 103
6.3.1 Définition des invariants d'architecture 103
6.3.2 FPATH : une syntaxe pour la navigation architecturale 104
6.3.3 Foreach : génération de motifs architecturaux pour FRACTAL ADL 106
6.3.4 Assert : vérification d'invariants architecturaux pour FRACTAL ADL 107
6.4 Illustration de l'utilisation des invariants architecturaux 108
6.4.1 Application à la définition du composant MySequence 108
6.4.2 Définition de motifs d'architecture génériques 110
6.5 Éléments d'implantation 113
6.5.1 Implantation de l'interpréteur FPATH 113
6.5.2 Extension de l'usine FRACTAL ADL 116
6.5.3 Extension du contrôle des composants FRACTAL 118
6.5.4 Extension des générateurs FRACLET 119
6.5.5 Discussion 120
6.6 Travaux connexes 120
6.6.1 Description de motifs d'architecture 121
6.6.2 Vérification d'invariants d'architecture 121
6.6.3 Intégration de nouvelles préoccupations 121
6.7 Conclusion 122

7 Canevas transactionnel GOTM 123
7.1 Introduction 124
7.2 Étude des points de variation 124
7.2.1 Adaptation du standard transactionnel 125
7.2.2 Adaptation du protocole de validation 125
7.2.3 Adaptation du modèle de transactions 126
7.2.4 Synthèse 126
7.3 Motivations 126
7.3.1 Flexibilité des services de transactions 126
7.3.2 Support hautement extensible pour l'adaptation 127
7.3.3 Granularité extrêmement fine de l'adaptation 127
7.4 Architecture du canevas GOTM 127
7.4.1 Modèle d'un service de transactions adaptable 127
7.4.2 Représentation dynamique d'un service de transactions 129
7.4.3 Architecture à base de composants d'un service de transactions 129
7.5 Éléments d'implantation du canevas GOTM 129
7.5.1 Implantation du motif de conception Adapter 131
7.5.2 Implantation des motifs de conception Factory et Prototype 132
7.5.3 Implantation du motif de conception Singleton 133
7.5.4 Implantation du motif de conception Strategy 134

7.5.5 Implantation du motif de conception State . . . 135
7.5.6 Implantation du motif de conception Command . . . 136
7.5.7 Implantation du motif de conception Publish-Subscribe . . . 138
7.6 Scénarii d'adaptation de GOTM . . . 138
7.6.1 Adaptation du protocole de validation de la transaction . . . 139
7.6.2 Adaptation des participants d'une transaction . . . 139
7.6.3 Adaptation d'un standard transactionnel . . . 139
7.7 Travaux connexes . . . 140
7.7.1 Canevas intergiciels . . . 140
7.7.2 Aspects et motifs de conception . . . 140
7.8 Conclusion . . . 141

8 Modèles dédiés à la configuration de GOTM **143**
8.1 Introduction . . . 144
8.2 Motivations et objectifs . . . 145
8.2.1 Motivations de l'ingénierie dirigée par les modèles . . . 145
8.2.2 Objectifs de l'application à la configuration des canevas intergiciels . . . 145
8.3 Modélisation du standard . . . 146
8.3.1 Modèle d'assemblage du standard transactionnel . . . 147
8.3.2 Modélisation du standard transactionnel JTS . . . 148
8.3.3 Éléments d'implantation . . . 149
8.4 Modélisation des états . . . 149
8.4.1 Présentation du diagramme UML d'états . . . 151
8.4.2 Modélisation de la machine à états d'une transaction . . . 152
8.4.3 Éléments d'implantation . . . 153
8.5 Modélisation des participants . . . 153
8.5.1 Présentation des règles Événement—Condition—Action . . . 154
8.5.2 Modélisation des participants de type *Synchronization* . . . 154
8.5.3 Éléments d'implantation . . . 156
8.6 Modélisation du protocole de validation . . . 156
8.6.1 Présentation du diagramme UML de séquences . . . 157
8.6.2 Modélisation du protocole de validation en deux phases . . . 157
8.6.3 Éléments d'implantation . . . 159
8.7 Conclusion . . . 159

III Expérimentations et validation **161**

9 Intégration de la démarcation transactionnelle **163**
9.1 Introduction . . . 164
9.2 Présentation de la démarcation transactionnelle . . . 165
9.2.1 Politiques de démarcation . . . 165
9.2.2 Domaines transactionnels . . . 166
9.3 Défis soulevés par l'étude de la démarcation de transactions . . . 166
9.3.1 Abstraction de la démarcation de transactions . . . 167
9.3.2 Abstraction du service de transactions . . . 167
9.3.3 Organisation des politiques de démarcation . . . 168
9.3.4 Intégration de nouvelles politiques de démarcation . . . 168
9.4 OTDF : le canevas de démarcation transactionnelle . . . 168
9.4.1 Présentation du canevas OTDF . . . 168
9.4.2 Abstraction du service de transactions . . . 169
9.4.3 Abstraction de la démarcation transactionnelle . . . 170
9.4.4 Intégration de nouvelles politiques . . . 172
9.5 JOTDF : l'implantation d'OTDF en Java . . . 173

9.5.1 Adaptateur de service de transactions . . . 173
9.5.2 Domaine transactionnel . . . 173
9.5.3 Politique transactionnelle . . . 173
9.5.4 Architecture d'un jeu de politiques transactionnelles . . . 175
9.5.5 Utilisation des politiques par les plates-formes applicatives . . . 175
9.6 Expérimentations . . . 175
9.6.1 Contexte et scénario . . . 177
9.6.2 Évaluation mémoire . . . 177
9.6.3 Évaluation de performances . . . 178
9.7 Conclusion . . . 178

10 Adaptation des standards transactionnels 179
10.1 Introduction . . . 180
10.2 Problématique de la composition . . . 180
10.3 Conception du service de transactions adapté . . . 181
10.3.1 Présentation du service de transactions adapté . . . 181
10.3.2 Analyse des fonctions . . . 181
10.3.3 Définition des stratégies . . . 185
10.3.4 Composition du service de transactions adapté . . . 187
10.3.5 Scénario d'utilisation du service de transactions . . . 188
10.4 Mise en œuvre et évaluation du service transactions adapté . . . 189
10.4.1 Éléments d'implantation avec GOTM . . . 189
10.4.2 Évaluation des performances . . . 190
10.5 Travaux connexes . . . 192
10.5.1 Interopérabilité des standards transactionnels . . . 192
10.5.2 Adaptation des services de transactions . . . 192
10.6 Conclusion . . . 193

11 Adaptation du protocole de validation 195
11.1 Introduction . . . 196
11.2 Présentation des protocoles de validation en deux phases . . . 196
11.2.1 Protocole *Two-Phase Commit* (2PC) . . . 197
11.2.2 Protocole *Two-Phase Commit Presumed Abort* (2PC-PA) . . . 197
11.2.3 Protocole *Two-Phase Commit Presumed Commit* (2PC-PC) . . . 198
11.3 Évaluation des protocoles de validation en deux phases . . . 199
11.3.1 Évaluation théorique . . . 199
11.3.2 Élements d'implantation . . . 199
11.3.3 Évaluation empirique . . . 201
11.3.4 Discussions . . . 203
11.4 CATE : un service de transactions sensible au contexte . . . 203
11.4.1 Sensibilité au contexte . . . 204
11.4.2 Règles de reconfiguration . . . 205
11.4.3 Politique d'adaptation du protocoles de validation en deux phases . . . 205
11.4.4 Évaluation empirique des performances de CATE . . . 206
11.4.5 Discussions . . . 207
11.5 Travaux connexes . . . 207
11.6 Conclusion . . . 208

12 Conclusion 209
12.1 Principaux apports . . . 209
12.1.1 Démarche extrêmement fine de construction de canevas intergiciels . . . 210
12.1.2 Canevas de construction de services de transactions hautement adaptables 211
12.2 Publications . . . 211
12.2.1 Revues internationales . . . 211

12.2.2 Conférences internationales 212
12.2.3 Ateliers internationaux 212
12.2.4 Conférence nationale 212
12.2.5 Atelier national 212
12.3 Perspectives 213
12.3.1 Axes de développement 213
12.3.2 Axes de recherche 213

Bibliographie **226**

Table des figures

1.1 Évolution des standards transactionnels dans le temps. 3
1.2 Évolution des protocoles de validation. 4
1.3 Démarche de construction d'intergiciels hautement adaptables. 8

2.1 Illustration de l'exécution d'une transaction. 15
2.2 Illustration du modèle de transactions emboîtées. 16
2.3 Illustration du modèle de transactions SAGAS. 17
2.4 Illustration du modèle de transactions coopérantes *split/join*. 17
2.5 Représentation du protocole de validation en 2 phases. 18
2.6 Représentation du protocole de validation en 2 phases décentralisé. 19
2.7 Représentation du protocole de validation en 2 phases hiérarchique. 19
2.8 Blocage dans l'exécution du protocole de validation en 2 phases. 20
2.9 Représentation du protocole de validation en 3 phases. 20
2.10 Représentation d'un service de transactions. 21
2.11 Représentation du standard DTP. 21
2.12 Représentation du standard OTS. 22
2.13 Représentation du standard JTS. 22
2.14 Représentation du standard WS-AT. 23
2.15 Architecture de *Reflective Transaction Framework*. 26
2.16 Architecture de l'approche KALA. 27
2.17 Architecture du canevas REFLECTS. 29
2.18 Architecture du service d'adaptation. 30
2.19 Abstraction fonctionnelle définies par ARJUNATS. 32

3.1 Positionnement du canevas intergiciel dans les applications modernes. 41
3.2 Exemples de personnalités de canevas intergiciels existants. 41
3.3 Architecture à base de composants FRACTAL. 42
3.4 Architecture de l'usine FRACTAL ADL. 44
3.5 Architecture à base de composants DREAM. 45
3.6 Architecture à base de composants OPENCOM. 46
3.7 Architecture à base de composants REMMOC. 49
3.8 Approche ABC centrée sur l'architecture. 49
3.9 Architecture à base de composants ABC. 50
3.10 Présentation du méta-modèle à 2 couches de PKUAS. 53

4.1 Illustration des objectifs d'un service de transactions hautement adaptable. . . . 64
4.2 Implantation de la fonction de journalisation en objet. 67
4.3 Implantation de la fonction de journalisation en composants de fine granularité. . 68
4.4 Représentation de la fonction de journalisation avec FRACTAL. 68

4.5 Présentation de la démarche GoTM de construction d'intergiciels. 71

5.1 Architecture FRACTAL du composant HelloWorld. 75
5.2 Modèle de programmation FRACLET. 79
5.3 Moteur de génération FRACLET. 80
5.4 Génération Java pour une application de type client-serveur. 81
5.5 Génération FRACTAL ADL pour une application de type client-serveur. 82
5.6 Architecture du moteur de génération FRACLET. 84
5.7 Illustration des générateurs de modèles de programmation spécifiques. 90
5.8 Description des composants OPENCOM HelloWorld. 91

6.1 Représentation graphique du composant Sequence. 101
6.2 Architecture de l'application HelloWorld. 104
6.3 Représentation graphique des chemins FPATH pour l'application HelloWorld. . . . 105
6.4 Représentation modulaire du composant MySequence. 110
6.5 Architecture de FPATH. 114
6.6 Architecture des chemins FPATH. 114
6.7 Architecture des axes FPATH. 115
6.8 Architecture des tests FPATH. 115
6.9 Architecture des fonctions FPATH. 116
6.10 Architecture des opérateurs FPATH. 117
6.11 Configurations de l'interpréteur FPATH. 117
6.12 Extension du composant Loader FRACTAL ADL. 118
6.13 Architecture des composants Foreach Loader et Assert Checker FRACTAL ADL. . 118
6.14 Intégration du contrôleur d'invariants à la membrane des composants FRACTAL. . 119

7.1 Adaptation des standards transactionnels dans GoTM. 125
7.2 Modèle d'un service de transactions adaptable. 128
7.3 Diagramme UML de séquences d'un service de transactions. 130
7.4 Architecture abstraite du service de transactions. 131
7.5 Architecture abstraite de la transaction. 131
7.6 Mise en œuvre du motif de conception Adapter dans GoTM. 131
7.7 Mise en œuvre des motifs de conception Factory et Prototype dans GoTM. 132
7.8 Mise en œuvre du motif de conception Singleton dans GoTM. 133
7.9 Mise en œuvre du motif de conception Strategy dans GoTM. 134
7.10 Mise en œuvre du motif de conception State dans GoTM. 135
7.11 Définition du composant regroupant les commandes de type `Synchronization`. 136
7.12 Mise en œuvre du motif de conception Command dans GoTM. 137
7.13 Mise en œuvre du motif de conception Publish-Subscribe dans GoTM. 138

8.1 L'ingénierie dirigée par les modèles. 146
8.2 Méta-modèle d'assemblage du standard transactionnel. 147
8.3 Architecture du composant associé au standard JTS. 149
8.4 Processus de construction d'un service de transactions. 150
8.5 Règles de transformation pour le diagramme UML d'états. 151
8.6 Diagramme UML d'états d'une transaction plate. 152
8.7 Architecture du composant à états State d'une transaction plate. 152
8.8 Modélisation et transformation d'un diagramme UML d'états. 153
8.9 Architecture du composant Synchronization Commands. 156
8.10 Transformation des règles en ECA en composants FRACTAL. 156
8.11 Diagramme de séquences UML. 157
8.12 Diagramme de séquences UML du protocole 2PC-PA. 158
8.13 Architecture du composant 2PC-PA. 159

9.1 Une organisation des politiques de démarcation transactionnelle. 166
9.2 Les défis liés à la démarcation transactionnelle. 167
9.3 Représentation du canevas logiciel OTDF. 169
9.4 Diagramme UML de classes de l'abstraction du service de transactions. 170
9.5 Diagramme UML de classes de l'abstraction de la démarcation transactionnelle. . . 171
9.6 Architecture à base de composants de la démarcation transactionnelle. 176
9.7 Diagramme UML de classes de l'abstraction de la démarcation transactionnelle. . . 176

10.1 Problème de la composition des standards transactionnels. 180
10.2 Illustration du service de transactions adapté. 182
10.3 Analyse des interfaces Java JTA. 183
10.4 Analyse des interfaces IDL CosTransactions. 184
10.5 Analyse des interfaces WSDL Atomic Transaction. 184
10.6 Présentation des stratégies disponibles. 185
10.7 Description de la stratégie *2PC*. 186
10.8 Description de la stratégie *Atomic Transaction State*. 187
10.9 Description de la composition du service de transactions adapté. 188
10.10 Description de la partie statique du service de transactions adapté. 189
10.11 Description de la partie dynamique du service de transactions adapté. 190
10.12 Passage à l'échelle du nombre de participants. 191
10.13 Passage à l'échelle du nombre de transactions. 191

11.1 Le protocole *Two-Phase Commit* (2PC). 197
11.2 Le protocole *Two-Phase Commit Presumed Abort* (2PC-PA). 198
11.3 Le protocole *Two-Phase Commit Presumed Commit* (2PC-PC). 198
11.4 Implantation à base de composants des protocoles 2PC. 200
11.5 Évaluation des protocoles pour un taux de validation important. 202
11.6 Évaluation des protocoles pour un taux d'abandon important. 202
11.7 Évaluation des protocoles pour un taux d'abandon unilatéral important. 203
11.8 Architecture de CATE. 204
11.9 Taux limite de reconfiguration du protocole de validation. 205
11.10 Architecture de la politique d'adaptation. 206
11.11 Évaluations des performances de CATE. 206

Liste des tableaux

2.1 Synthèse de l'adaptabilité dans les intergiciels transactionnels. 35

3.1 Description du langage FRACTAL ADL. 43
3.2 Synthèse des modèles de composants. 57

5.1 Jeu d'annotations pour le modèle de programmation de FRACTAL. 79
5.2 Mesures empiriques du code source réalisées sur l'exemple HelloWorld. 87
5.3 Mesures empiriques du code source réalisées sur le serveur Web Comanche. 88
5.4 Mesures empiriques du code généré réalisées sur l'application HelloWorld. 88
5.5 Mesures empiriques du code généré réalisées sur le serveur Web Comanche. . . . 88
5.6 Mesures empiriques du code généré réalisées sur le projet TMF. 89
5.7 Mesures empiriques du code généré réalisées sur le projet GOTM. 89
5.8 Mesures empiriques du code généré réalisées sur le projet FDF. 89

6.1 Description des axes FPATH. 105

7.1 Politiques du composant Singleton. 133

8.1 Jeu d'annotations pour le modèle de programmation de FRACTAL. 154

9.1 Politiques de démarcation définie par CCM et EJB 165
9.2 Évaluation de la taille de JOTDF. 177
9.3 Évaluation du coût d'exécution des politiques de JOTDF. 178

11.1 Coût théorique des protocoles 2PC. 199

Liste des listings

3.1 Description FRACTAL ADL du composant Application. 43
3.2 Description ACME/*Armani* du type d'une pile de protocoles. 47
3.3 Description PLASTIK d'une pile de protocoles. 48
3.4 Description ABC/ADL d'un composant et d'un connecteur. 51
3.5 Description ABC/ADL d'un composant et d'un connecteur. 52
3.6 Description ARCHJAVA d'un composant primitif analyseur lexical. 54
3.7 Description ARCHJAVA de l'assemblage d'un compilateur. 54
3.8 Description DARWIN d'une chaîne de délégation. 55
4.1 Description FRACTAL ADL du composant Logging. 69
5.1 Description de l'interface Service. 75
5.2 Mise en œuvre du composant FRACTAL Client. 75
5.3 Description de l'interface de contrôle FRACTAL ServerAttributes. 76
5.4 Mise en œuvre du composant FRACTAL Server. 76
5.5 Description FRACTAL ADL du composant Client. 76
5.6 Description FRACTAL ADL du composant Server. 77
5.7 Description FRACTAL ADL du composant HelloWorld. 77
5.8 Description annotée de l'interface Service. 82
5.9 Mise en œuvre annotée du composant Client. 82
5.10 Mise en œuvre annotée du composant Server. 83
5.11 Description FRACTAL ADL du composant HelloWorld. 83
5.12 Code source généré pour l'attribut header. 84
5.13 Code source généré pour le composant Client. 85
5.14 Description FRACTAL ADL générée pour l'interface Service. 85
5.15 Code source généré pour le composant Server. 85
5.16 Description FRACTAL ADL générée pour le composant Client. 86
5.17 Description FRACTAL ADL générée pour le composant Server. 86
5.18 Description FRACTAL ADL générée pour le composite Client. 86
5.19 Composant OPENCOM Client. 92
5.20 Composant OPENCOM Server. 93
6.1 Description FRACTAL ADL du composant MySequence. 102
6.2 Description de la syntaxe de l'opérateur `foreach`. 106
6.3 Description FRACTAL ADL du composant MySequence. 107
6.4 Description de la syntaxe de l'opérateur `assert`. 107
6.5 Description FRACTAL ADL du composant MySequence. 108
6.6 Description FRACTAL ADL de l'intégration du composant Logger. 108
6.7 Description FRACTAL ADL générique d'une séquence de composants. 109
6.8 Description FRACTAL ADL de l'intégration du composant SI. 109
6.9 Intégration des motifs Sequence, SI et AutoLogger dans une séquence d'étapes. . . . 109
6.10 Motif FRACTAL ADL de l'exportation d'une interface serveur collection. 111

6.11 Motif FRACTAL ADL de l'exportation des interfaces serveurs d'un composant. . . . 111
6.12 Motif FRACTAL ADL de l'importation d'une interface cliente. 112
6.13 Motif FRACTAL ADL du partage de composants. 112
6.14 Motif FRACTAL ADL de la liaison d'une interface cliente collection. 113
6.15 Description FRACTAL ADL du composant Backend. 115
6.16 Description des interfaces de contrôle des invariants. 119
6.17 Description FRACTAL ADL de la description générée ClientAutoBind. 120
7.1 Description FRACTAL ADL des composants StateMachine et State. 136
7.2 Description FRACTAL ADL du composant SynchronizationCommands. 137
7.3 Implantation des commandes de *Synchronization* dans GOTM. 137
7.4 Adaptation du protocole de validation. 139
7.5 Adaptation d'un service de transactions de type JTS. 140
8.1 Description d'un service de transactions de type JTS. 148
8.2 Description FRACTAL ADL d'un service de transactions de type JTS. 149
8.3 Modélisation ECA des participants de type *Synchronization*. 155
8.4 Description FRACTAL ADL du composant SynchronizationCommands. 155
9.1 Adaptateur du service de transactions JTS. 174
9.2 Illustration du domaine d'interruption. 174
9.3 Illustration de la politique NotSupported. 174
9.4 Description FRACTAL ADL des composants Domains et Policies. 175
10.1 Description des participants de type *Synchronization*. 186

New systems generate new problems.

Murphy's Law

Chapitre 1

Introduction

Sommaire

1.1 Contexte du travail **1**
1.2 Problématique **2**
1.2.1 Axes d'évolution des services de transactions 2
1.2.2 Problématique de la construction d'intergiciels hautement adaptables . . 5
1.3 Proposition **6**
1.3.1 Granularité de l'adaptation 6
1.3.2 Contributions à la conception des services de transactions 8
1.3.3 Validation des services de transactions construits avec GoTM 9
1.4 Organisation du document **9**
1.4.1 État de l'art 9
1.4.2 Contributions 9
1.4.3 Validation 10

CE CHAPITRE INTRODUCTIF POSE LES BASES DU TRAVAIL RÉALISÉ DANS CETTE THÈSE. La section 1.1 présente le contexte dans lequel s'est réalisée cette thèse. La section 1.2 introduit la problématique de l'adaptabilité des services intergiciels, et plus particulièrement des services de transactions, adressée dans ce travail. Puis, la section 1.3 détaille notre proposition d'une démarche à granularité extrêmement fine pour la construction de canevas intergiciels et notre canevas dédié à la construction des services de transactions hautement adaptables. Finalement, la section 1.4 présente l'organisation en trois parties de ce document.

1.1 Contexte du travail

Ce travail a été réalisé au sein du projet JACQUARD de l'INRIA et de l'équipe GOAL du Laboratoire d'Informatique Fondamentale de Lille (LIFL). La problématique du projet JACQUARD adresse la conception d'applications réparties complexes. Ces applications sont typiquement composées d'un nombre important de composants distribués sur un réseau. Dans ce contexte, l'objectif du projet JACQUARD est de produire de nouvelles plates-formes, outils, méthodologies et approches pour maîtriser et faciliter la conception de cette catégorie d'applications. En particulier, le projet explore les modèles de composants, la séparation des préoccupations et le tissage à différentes étapes du cycle de vie d'une application, c.-à-d. la modélisation, la conception, l'assemblage, le déploiement et l'exécution. La validation des travaux du projet JACQUARD passe par sa participation active dans des organismes de standardisation tels que l'*Object Management Group* (OMG) ou le consortium international pour la promotion des développements libres *ObjectWeb*.

Les travaux de cette thèse s'inscrivent pleinement dans la thématique du projet JACQUARD et contribuent à la problématique du support de l'adaptabilité dans les plates-formes intergicielles.

1.2 Problématique

De nos jours, la majorité des applications distribuées s'exécutent sur des plates-formes intergicielles. Ces plates-formes jouent le rôle de médiateur entre le système d'exploitation et l'application. Elles fournissent des abstractions fonctionnelles du système d'exploitation à l'application en masquant leur hétérogénéité et leur complexité. Parmi ces abstractions, nous pouvons citer les fonctions de communication, de persistance, de sécurité ou encore de transaction. Une transaction correspond à l'exécution d'une séquence d'opérations dont les effets doivent être globalement appliqués ou annulés le cas échéant. Le support de cette fonction transactionnnelle est généralement assurée par un service intergiciel dédié. Le rôle de ce service de transactions consiste alors à isoler une séquence d'opérations afin de prendre en charge la coordination de leurs effets lors de la terminaison de la transaction.

Cependant, les plates-formes intergicielles doivent évoluer continuellement pour prendre en compte de nouveaux besoins. Ces évolutions correspondent à des améliorations des paradigmes d'une technologie existante comme par exemple le passage de la technologie Java RMI (*Remote Method Invocation*) [GJSB05] associée au paradigme de conception par objets [DEMN98] à la technologie J2EE *Java 2 Enterprise Edition* [DeM03] associée au paradigme de conception par composants [SGM02]. Mais elles peuvent également correspondre à un changement brutal de technologie tel que le passage du standard CORBA (*Common Object Request Broker Architecture*) [OMG04] au standard J2EE et plus récemment aux *Web Services* [GHM+03, CMRW06]. Toutes ces évolutions requièrent non seulement une adaptation des plates-formes intergicielles supportant ces différents standards mais également une adaptation plus profonde des différentes fonctions intégrées dans les plates-formes. Cependant, des modifications trop profondes des plates-formes intergicielles peuvent constituer un frein à l'adaptation de cette dernière. **Dès lors, nous pensons qu'il est nécessaire de s'interroger sur la façon de construire un intergiciel capable de s'adapter aux différentes évolutions qu'il peut rencontrer.**

Pour illustrer cette nécessité de forte adaptation des intergiciels, nous proposons de nous focaliser sur la construction d'un service de transactions. Le service de transactions nous semble particulièrement intéressant car les différents axes d'évolution auxquels il peut être confronté représentent des évolutions adressées par les plates-formes intergicielles.

1.2.1 Axes d'évolution des services de transactions

En étudiant les services de transactions, nous souhaitons nous concentrer sur l'évolution de l'intégration de la fonction transactionnelle dans les plates-formes intergicielles mais aussi sur l'évolution des caractéristiques des services de transactions. Parmi ces caractéristiques, les standards transactionnels, les protocoles de validation et les modèles de transactions ont fait l'objet de nombreuses évolutions durant ces dernières décennies.

Axe des standards transactionnels

Les standards transactionnels garantissent la compatibilité d'un service de transactions avec la plate-forme intergicielle à laquelle il peut être associé. Pour ce faire, la spécification d'un standard transactionnel comporte notamment la définition d'un jeu d'interfaces afin d'assurer l'intégration du service implantant ce standard dans une plate-forme intergicielle. Mais cette spécification peut également imposer le modèle de transactions et le protocole de validation que le service de transactions doit employer. Il existe de nombreux standards transactionnels tels que *Distributed Transaction Processing* (DTP) [The96], *Object Transaction Service* (OTS) [OMG03], *Java Transaction Service* (JTS) [Che99] ou *Web Services Atomic Transaction* (WS-AT) [CCF+05a]. Cependant, la liste de ces standards ne peut être définitive car l'évolution de leurs spécifications

est intimement liée à l'évolution des technologies sur lesquelles ils reposent. Ainsi, le standard DTP est apparu dans les premiers systèmes modulaires. Puis, l'émergence des systèmes à objets distribués et de la technologie CORBA a permis de définir le standard OTS. Ce standard a ensuite été décliné pour le langage Java avec JTS lorsque la technologie J2EE a fait son apparition. Plus récemment, c'est le succès de la technologie *Web Services* qui a encouragé la définition du standard WS-AT. Il est donc logiquement envisageable que les prochaines évolutions des technologies de conception logicielle entraîneront inévitablement la définition de nouveaux standards transactionnels.

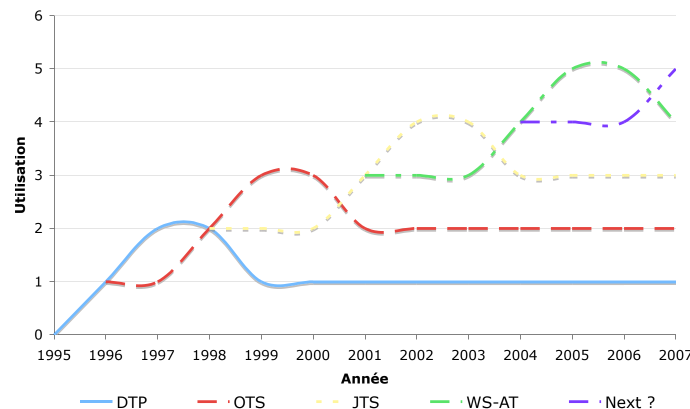


FIG. 1.1: Évolution des standards transactionnels dans le temps.

La figure 1.1 illustre l'évolution de ces différents standards transactionnels dans le temps. Outre le fait que les standards transactionnels rencontrent tous une période de fort succès à laquelle succède un léger déclin, il est surtout intéressant d'observer deux éléments. Tout d'abord, la définition des standards transactionnels est incrémentale afin de garantir une compatibilité ascendante de la fonction transactionnelle. En effet, les fonctionnalités spécifiées par un nouveau standard englobent généralement les fonctionnalités définies par le standard précédent. Ensuite, les standards transactionnels en déclin ne disparaissent pas mais ils coexistent avec les standards plus récents. Notamment, les standards DTP, OTS, JTS et WS-AT demeurent une référence dans leur contexte d'utilisation et permettent aux applications patrimoniales de conserver un support de la fonction transactionnelle.

Ces aspects de compatibilité incrémentale et de coexistance des standards transactionnels impactent par conséquent le développement des services de transactions. Cependant, **peu de services de transactions existants prennent en compte ces deux aspects liés à l'évolution des standards transactionnels**. Bien souvent, les services de transactions se contentent de suivre l'évolution des standards en implantant les nouvelles spécifications sans se soucier du devenir des précédents standards. Ce choix se justifie par le fait qu'il est généralement difficile de maîtriser l'accumulation des standards transactionnels sans que les performances du service de transactions ne s'en trouvent impactées. De plus, la multiplication des standards supportés par un service de transactions a tendance à restreindre ses capacités d'adaptation en spécialisant ses fonctionnalités. **Peu de services définissent leur architecture comme une structure suffisamment modulaire pour anticiper les évolutions futures**.

Axe des protocoles de validation

Les protocoles de validation d'une transaction ont également subi de nombreuses évolutions. À l'origine, ce protocole ne comportait qu'une seule phase de traitement car il était employé dans des systèmes centralisés. Cependant, l'émergence des systèmes distribués a nécessité la définition d'une algorithmique prenant en compte les aspects de répartition et de tolérance aux fautes des

ressources transactionnelles. La ressource transactionnelle représente l'abstraction d'une donnée modifiable par une transaction dans un système réparti. Dès lors, le protocole de validation en deux phases (*Two-Phase Commit* [2PC]) a fait son apparition. Ce protocole permet d'établir un consensus quant à l'issue d'une transaction entre plusieurs ressources transactionnelles réparties sur le réseau. Cependant, ce protocole n'est pas parfait et son utilisation peut s'avérer limitée dans certains contextes d'exécution. Dès lors, des variantes et des optimisations du protocole 2PC ont permis d'améliorer ses performances dans des contextes déterminées. Par exemple, le protocole 2PC-PA (*2PC Presumed Abort*) optimise le coût de validation des transactions ayant tendance à abandonner alors que le protocole 2PC-PC (*2PC Presumed Commit*) minimise le coût de transactions présetant un taux de validation élevé. Mais ces variantes peuvent également permettre de considérer des transactions aux sémantiques particulières comme les transactions en lecture seule ou supporter des topologies de réseaux originales telles que les réseaux mobiles.

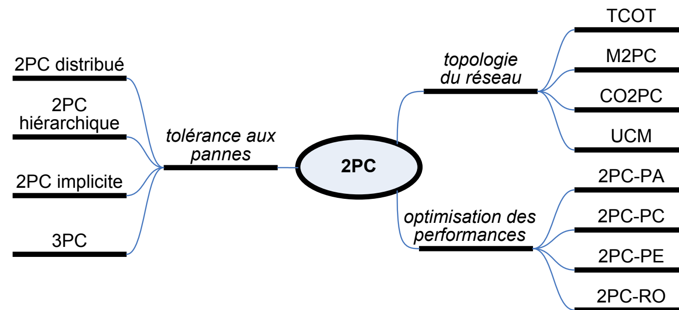


FIG. 1.2: Évolution des protocoles de validation.

La figure 1.2 présente différentes évolutions des protocoles de validation. Il est intéressant de noter que les évolutions des protocoles ont été orientées selon des préoccupations différentes telles que le support de la *tolérance aux fautes*, l'*optimisation des performances* ou le support des *topologies de réseaux*. Ainsi, les protocoles 3PC (*Three-Phase Commit*), 2PC distribué, 2PC hiérarchique et 2PC implicite sont des exemples de protocoles de validation cherchant à renforcer la fiabilité du processus de validation en envisageant différents types de fautes. Les protocoles TCOT (*Timeout-based Mobile Transaction Commitment*), M2PC (*Mobile 2PC*), CO2PC (*Combination of an Optimistic approach and 2PC*) et UCM (*Unilateral Commit for Mobile Environment*) proposent différentes solutions pour le support des déconnexions dans les environnements mobiles [SARA04]. Enfin, les protocoles 2PC-PA, 2PC-PC, 2PC-PE (*2PC Presumed Either*) et 2PC-RO (*2PC Read Only*) définissent des optimisations du protocole 2PC pour des sémantiques d'exécution particulières telles que les transactions abandonnant ou validant majoritairement ou les transactions ne réalisant que des lectures [GR93].

Dès lors, il n'existe pas de protocole de validation performant adapté à tous les contextes d'exécution. De plus, **peu de services de transactions existants prennent en compte ces différentes variantes des protocoles de validation**. Généralement, les services de transactions configurent leur protocole de validation de manière statique sans prendre en compte les aspects d'optimisation liés au contexte d'exécution. Ce protocole de validation est conservé tout au long de l'exécution du service de transactions et ses performances sont alors fortement dépendantes du contexte d'exécution dans lequel il est employé. La difficulté du choix du protocole de validation repose sur la connaissance *a priori* du contexte d'exécution du service de transactions afin de choisir le protocole le plus approprié.

Axe des modèles de transactions

Les modèles de transactions correspondent à un autre aspect d'évolution des services de transactions. Le modèle de transactions le plus employé est le modèle des transactions plates.

Ce modèle est utilisé par les bases de données pour isoler les accès concurrents à un ensemble d'enregistrements. Il est particulièrement adapté pour les transactions s'exécutant sur un intervalle de temps relativement court. Cependant, de plus en plus d'applications utilisent des transactions pour isoler des activités s'exécutant sur le long terme. Dans ces conditions, le modèle de transactions plates atteint ses limites car il verrouille l'accès aux données modifiées durant toute l'exécution de la transaction. Sur le long terme, ce verrouillage handicape le fonctionnement de l'application et peut conduire à des situations d'interblocage. Dès lors, de nouveaux modèles de transactions sont apparus afin de permettre à des transactions de s'exécuter sur le long terme sans que cela n'impacte néfastemement les performances de l'application. Ces modèles étendus permettent de relâcher plus rapidement les verrous associés aux données modifiées en découpant une transaction monolithique comme une séquence ou une hiérarchie de sous-transactions s'exécutant sur une durée plus courte. Ces modèles se chargent ensuite de définir la sémantique des sous-transactions et de coordonner leur exécution. Les modèles de transactions SAGAS, *Split/Join* ou *Nested Transactions* (NT) représentent différentes versions de ces modèles de transactions étendus [JK97]. Leurs principales différences portent sur la gestion des dépendances entre les sous-transactions. Ainsi, le modèle SAGAS organise les sous-transactions comme une séquence de transactions dont les effets peuvent être compensés par d'autres transactions en cas d'abandon. Le modèle *Split/Join* permet de diviser et de fusionner des transactions indépendantes afin de pouvoir accéder aux données qu'elles modifient. Enfin, le modèle NT se comporte comme une hiérarchie de sous-transactions dont les effets sont visibles en fonction des relations de parenté.

Afin de maîtriser l'évolution des modèles de transactions, le formalisme ACTA permet de caractériser un modèle de transactions en utilisant des concepts de visibilité, de conflit, de délégation et de dépendance. Cette formalisation des modèles de transactions offre un mécanisme unique de modélisation des transactions. Des travaux récents [Fab05, Bar98] ont permis de reproduire ces concepts au niveau du service de transactions afin que celui-ci puisse supporter différents modèles de transactions décrits à partir d'une spécification ACTA mais **les performances de ces services sont bien souvent plus mauvaises que les services de transactions n'implantant qu'un seul modèle de transactions**.

1.2.2 Problématique de la construction d'intergiciels hautement adaptables

Face à ces différentes évolutions des services de transactions, les travaux existants se sont concentrés sur l'adaptation d'un seul axe d'évolution tel que le standard transactionnel, le protocole de validation ou le modèle de transactions. Les évolutions supportées dans ces approches peuvent être adressées par une adaptation lors de la conception ou lors de l'exécution du service de transactions. Certes, ces travaux fournissent des réponses pertinentes pour chacune de ces évolutions, mais ils présentent également certaines limites quant au support des autres axes d'évolutions. **Notamment, aucune proposition n'est en mesure d'adresser simultanément plusieurs axes d'évolution au sein d'un même service de transactions**.

Ce constat que nous faisons quant à l'adaptabilité des services de transactions est généralisable aux autres services intergiciels car chacun d'eux est soumis à l'évolution de standards et de modèles spécifiques aux fonctions qu'il réalise.

Par conséquent, nous pensons que la réalisation d'un intergiciel hautement adaptable nécessite de répondre préalablement aux cinq questions suivantes :

- *Quelle approche faut-il adopter pour construire l'intergiciel ?*
- *Quel paradigme peut être utilisé pour concevoir l'intergiciel ?*
- *Quelle granularité d'adaptation de l'intergiciel faut-il adresser ?*
- *Quel type d'adaptation peut être supporté par l'intergiciel ?*
- *Comment préserver les performances des intergiciels ?*

1.3 Proposition

L'objectif de cette thèse est de proposer une solution suffisamment modulaire pour prendre en compte l'intégralité des évolutions qu'un intergiciel peut rencontrer. Nous souhaitons pouvoir adapter l'intergiciel **lors de sa conception ou lors de son exécution** à **des évolutions de n'importe quelle nature**.

Nous choisissons d'appliquer notre proposition à la construction de services de transactions hautement adaptables. Nous adoptons l'**approche d'un canevas intergiciel dédié à la construction de services de transactions hautement adaptables** car elle permet de construire différentes personnalités de services de transactions en fonction des axes d'évolution que nous souhaitons adresser. Ensuite, nous choisissons d'utiliser le **paradigme de conception par composants** afin de bénéficier de ses propriétés de modularité. En particulier, nous pensons que le modèle de composants FRACTAL [BCL+06] est un bon candidat pour la conception de services intergiciels hautement adaptables.

1.3.1 Granularité de l'adaptation

Les différentes adaptations d'un intergiciel peuvent adresser des granularités différentes du service de transactions. Si le choix du standard transactionnel impacte principalement le service de transactions, le protocole de validation modifie le comportement des transactions manipulées par ce service. Quant au choix du modèle de transactions, il peut opérer des modifications à la granularité du service de transactions et à celle des transactions. Par conséquent, cette différence de granularité d'adaptation du service de transactions requiert une modélisation très fine d'un service de transactions afin de pouvoir intervenir à n'importe quelle granularité. Notre canevas intergiciel GOTM (GOTM *is an open Transaction Monitor*) propose donc de construire les fonctionnalités d'un service de transactions comme une bibliothèque de composants de très fine granularité.

Vers des services de transactions hautement adaptables

À titre d'exemple, les travaux réalisés récemment autour du serveur d'applications JONASÀLACARTE [Abd06] ont montré l'apport d'une démarche de conception à base de composants FRACTAL dans la structuration d'une plate-forme intergicielle. Grâce à cette approche, la configuration, le déploiement et l'administration du serveur d'applications est simplifiée par la réification de l'architecture sous la forme d'un assemblage de composants. Cette approche définit une dizaine de composants utilisés pour structurer l'architecture de la plate-forme intergicielle. En particulier, le service de transactions du serveur d'applications JONASÀLACARTE est représenté par un composant monolithique. Cependant, cette granularité est trop grosse pour pouvoir supporter l'évolution des différentes caractéristiques d'un service de transactions. C'est pour cette raison que nous proposons de poursuivre cette démarche de décomposition fonctionnelle de la plate-forme intergicielle à la modélisation du service de transactions.

Notre démarche vise ainsi à concevoir un service de transactions comme un composant modulaire intégrable dans n'importe quelle plate-forme intergicielle. Ce service de transactions contrôle un ensemble de transactions en cours d'exécution, elles-même représentées comme des composants modulaires. Le cycle de vie de chacune de ces transactions, c.-à-d. transaction en cours, validée, abandonnée, est matérialisé par un composant composite dont les différents états sont eux-mêmes représentés par des composants primitifs.

Vers une démarche de conception à granularité extrêmement fine

La modélisation d'un service de transactions en utilisant des composants de très fine granularité implique une augmentation importante du nombre des composants impliqués dans la réalisation du service de transactions. Dès lors, la taille et le nombre des composants de très fine granularité pose de nouveaux problèmes quant à la conception du service de transactions :

– le *coût du développement d'un composant de très fine granularité* doit être réduit par rapport à une approche traditionnelle à plus gros grain ;
– la *réutilisabilité des composants de très fine granularité* et des constructions élémentaires doit être maximisée afin d'anticiper sur les différents degrés d'adaptabilité du service de transactions ;
– la *complexité de la description des assemblages de composants de très fine granularité* et le contrôle de leur cohérence doivent être renforcés pour limiter les erreurs lors des phases de configuration et de reconfiguration du système ;
– le *support et l'aide à l'assemblage de composants de très fine granularité* doivent permettre de masquer la complexité de la réalisation du service de transactions pour se concentrer sur la configuration des caractéristiques de ce dernier.

Face aux problèmes liés à l'utilisation d'une granularité extrêmement fine de décomposition d'un service de transactions en de nombreux composants atomiques, nous proposons de définir une **démarche de conception à granularité extrêmement fine** afin de maîtriser la construction d'un intergiciel hautement adaptable.

Identification des acteurs de conception. Nous proposons d'identifier trois rôles afin de structurer notre démarche de conception. Le rôle d'**intégrateur** nous permet de nous concentrer sur la modélisation d'un service de transactions hautemement adaptable. L'intégrateur assure le support et l'aide à l'assemblage des composants de très fine granularité en manipulant des modèles de haut niveau dédiés à la configuration des caractéristiques d'un service de transactions.

Le rôle d'**architecte** nous permet de nous préoccuper de la conception d'un service de transactions comme un assemblage de composants de très fine granularité. L'architecte maîtrise la complexité et la cohérence de la description des assemblages des composants de très fine granularité en définissant des motifs d'architecture dont la validité peut être vérifiée au déploiement et à l'exécution du service de transactions. L'expression de ces motifs est faite via un langage de description dédié aux motifs d'architecture.

Enfin, le rôle de **développeur** nous permet de nous focaliser sur le développement des fonctionnalités de notre canevas intergiciel GOTM. Le développeur optimise le coût de développement et la réutilisabilité des composants de très fine granularité en s'appuyant sur un modèle de programmation fiable et efficace.

Intégration d'un service de transactions. L'intégration d'un service de transactions construit avec GOTM dans une plate-forme intergicielle requiert de définir non seulement le standard transactionnel à respecter mais aussi l'algorithme du protocole de validation et le modèle de transactions. Nous proposons donc d'employer des modèles de haut niveau pour décrire les caractéristiques du service de transactions. Ces modèles font abstraction de la conception du service de transactions et permettent de décrire le comportement des différentes préoccupations dans un formalisme adapté. Ces modèles sont ensuite interprétés et automatiquement transformés en un assemblage de composants de très fine granualité réalisant le service de transactions modélisé.

Architecture d'un service de transactions. L'architecture d'un service de transactions construit à partir de GOTM est composée d'un très grand nombre de composants de très fine granularité. La présence d'un grand nombre de composants dans une architecture entraîne automatiquement une augmentation du risque d'erreurs de description de l'assemblage. Cependant, ce type d'architecture présente également la particularité de faire apparaître un certain nombre de motifs récurrents dans l'assemblage des composants. Ces motifs d'architecture peuvent être identifiés et isolés à l'aide d'un langage dédié afin de systématiser et de contrôler leur application dans une architecture. La description de ces motifs permet de faire remonter différents patrons de conception au niveau de l'architecture du service de transactions. La description concrète d'un service de transactions est alors réduite à la déclaration des composants de très fine granularité nécessaires pour implanter les différentes caractéristiques du service de transactions. La déclaration de ces composants est obtenue par transformation des modèles de haut niveau définis par l'intégrateur.

Développement du canevas intergiciel GoTM. Le développement du canevas intergiciel GoTM nécessite de définir un modèle de programmation adapté à la granularité des composants réalisant le service de transactions. Notre modèle de programmation, appelé FRACLET, est ensuite utilisé par le développeur pour réaliser les fonctionnalités fournies par le canevas intergiciel GoTM. L'objectif de ce modèle est de minimiser l'impact du modèle de composants utilisé pour développer les composants de très fine granularité afin de (1) faciliter et fiabiliser le développement et (2) isoler la réalisation des fonctionnalités et la technologie de mise en œuvre. Les composants de GoTM sont facilement maintenables et la projection vers un modèle de composants déterminé est automatisée par une processus de génération.

1.3.2 Contributions à la conception des services de transactions

Les contributions présentées dans cette thèse visent à fournir une démarche de construction des services de transactions hautement adaptables en utilisant une granularité de conception extrêmement fine. Pour ce faire, nous proposons les contributions suivantes :

1. une **démarche de conception structurée selon trois rôles**,
2. un **modèle de programmation**, nommé FRACLET,
3. un **langage de description et de vérification de motifs d'architecture**,
4. un **canevas intergiciel**, nommé GoTM,
5. des **modèles de haut niveau de configuration de GoTM**.

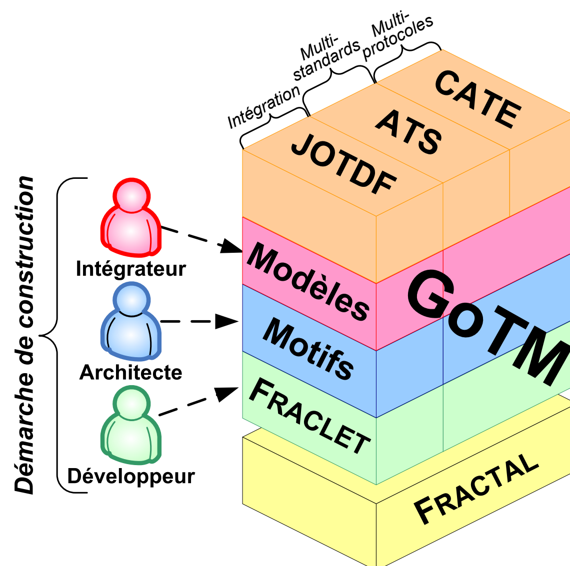


FIG. 1.3: Démarche de construction d'intergiciels hautement adaptables.

La figure 1.3 positionne nos contributions dans la définition de la démarche de construction des intergiciels hautemement adaptables. Le développeur utilise notre modèle de programmation FRACLET pour développer les composants élémentaires du canevas GoTM. FRACLET propose d'appliquer les principes de la programmation par attributs pour simplifier la programmation des composants FRACTAL en remplaçant les aspects techniques d'un composant FRACTAL par des annotations. L'architecte utilise notre langage de description et de vérification de **motifs** d'architecture pour décrire les motifs de conception utilisés dans le canevas GoTM. Ce langage permet d'identifier, d'appliquer et de vérifier les motifs de construction récurrentes d'une architecture à base de composants en combinant le langage FPATH [Dav05] avec deux nouveaux opérateurs du langage FRACTAL ADL. Enfin, l'intégrateur utilise nos **modèles** de haut niveau pour décrire les configurations des services de transactions construits à partir du canevas GoTM.

1.3.3 Validation des services de transactions construits avec GoTM

La validation de notre démarche est réalisée au travers de trois expériences originales que nous avons illustré dans la figure 1.3.

Pour supporter l'évolution des standards transactionnels, nous présentons le service de transactions **ATS**, construit à partir du canevas intergiciel GoTM, dont la caractéristique principale est d'implanter plusieurs standards transactionnels simultanément. Grâce à ce service, il est possible d'assurer l'interopérabilité transactionnelle d'applications patrimoniales basées sur des technologies hétérogènes (CORBA, J2EE, *Web Services*).

Ensuite, pour supporter l'évolution des protocoles de validation, nous présentons le service de transactions **CATE**. Celui-ci est construit également avec les composants du canevas GoTM et il est capable d'adapter dynamiquement le protocole de validation d'une transaction au contexte d'exécution afin de minimiser le délai de validation de cette transaction. Grâce à ce service, le choix du protocole de validation ne nécessite pas une connaissance *a priori* du contexte d'exécution car il est automatiquement déterminé et pris en charge par le service de transactions.

Enfin, pour supporter l'évolution de l'intégration des fonctionnalités transactionnelles requises par les plates-formes intergicielles, nous proposons d'isoler la fonction de démarcation transactionnelle dans le composant modulaire **JOTDF**, indépendant de la plate-forme intergicielle cible et du service de transactions sous-jacent. Ce composant fournit alors un support extensible de la fonction de démarcation transactionnelle.

1.4 Organisation du document

La suite de ce document est organisée en trois parties. La première partie se concentre sur l'état de l'art relatif à nos travaux. La deuxième partie détaille nos contributions à la démarche de construction de canevas intergiciels hautement adaptables. Enfin, la troisième partie présente les trois expériences que nous avons réalisées pour valider notre proposition.

1.4.1 État de l'art

La première partie de ce document est dédiée à l'état de l'art des principaux domaines abordés dans cette thèse. Le chapitre 2 présente tout d'abord le domaine transactionnel. Il y introduit les notions de base nécessaires à la compréhension de cette thèse, puis il étudie différents travaux adressant sous diverses formes la préoccupation d'adaptabilité dans ce domaine. Le chapitre 3 présente les différents aspects du principe de séparation des préoccupations et se concentre ensuite sur les différents modèles de composants existants dans le domaine de la construction des intergiciels. L'objectif de cette partie est de situer le domaine d'application de nos travaux et de déterminer les forces et les faiblesses des approches transactionelles actuelles. Ces forces et ces faiblesses combinées à la problèmatique de cette thèse motivent la deuxième partie de la thèse.

1.4.2 Contributions

La deuxième partie du document présente les contributions de cette thèse. Le chapitre 4 introduit les quatre éléments et les trois rôles constituant le cœur de notre proposition. Ces différents éléments s'intègrent dans notre démarche de construction de services de transactions hautement adaptables. Le chapitre 5 présente notre modèle de programmation des composants FRACTAL nommé FRACLET. Le chapitre 6 présente notre langage de description et de vérification de motifs d'architecture. Le chapitre 7 présente la bibliothèque de composants transactionnels GoTM. Ce canevas définit une architecture abstraite d'un service de transactions en utilisant des motifs de conception reconnus qu'il réalise ensuite par des assemblages de composants de très fine granularité. Enfin, le chapitre 8 introduit les quatre modèles de configuration que nous avons définis pour configurer le canevas logiciel GoTM. Ces modèles de haut niveau sont convertis automati-

quement vers des assemblages de composants du canevas GOTM comme nous l'illustrons dans la troisième partie.

1.4.3 Validation

La troisième partie du document correspond aux expériences que nous avons réalisées pour valider notre démarche de construction. Le chapitre 9 présente l'abstraction de la démarcation transactionnelle dans les plates-formes à base de composants. Cette abstraction, nommée **JOTDF**, permet d'intégrer les six politiques de démarcation généralement définies par les spécifications des modèles de composants dans différentes plates-formes à base de composants. Le chapitre 10 présente la composition de plusieurs standards transactionnels pour assurer l'interopérabilité transactionnelle de plusieurs applications patrimoniales hétérogènes. Cette composition est réalisée par le service de transactions **ATS** qui implante plusieurs standards transactionnels simultanément. Enfin, le chapitre 11 présente l'adaptation dynamique du protocole de validation en deux phases d'un service de transactions. Cette adaptabilité est supportée par le service de transactions **CATE** capable de reconfigurer le protocole de validation en observant le contexte d'exécution.

Première partie

État de l'art

Any given program, when running, is obsolete.

Murphy's Law

Chapitre 2

Étude de l'adaptabilité dans les intergiciels transactionnels

Sommaire

2.1 Introduction **14**
2.2 Prérequis **14**
2.2.1 Définition d'une transaction 14
2.2.2 Propriétés ACID 14
2.2.3 Modèles de transactions 15
2.2.4 Atomicité globale 17
2.2.5 Services de transactions 20
2.2.6 Standards transactionnels 21
2.2.7 Synthèse 23
2.3 Étude de l'existant **23**
2.3.1 Critères d'évaluation 24
2.3.2 *Reflective Transaction Framework* 25
2.3.3 *Kernel Aspect Language for ATMS* 26
2.3.4 ReflecTS 28
2.3.5 Service d'adaptation des services techniques 30
2.3.6 *Arjuna Transaction System* 31
2.3.7 Autres approches 32
2.4 Synthèse **33**
2.5 Conclusion **36**

Ce chapitre présente une étude du domaine transactionnel. *Cette étude se compose d'une présentation de la notion de transaction et de ses différentes évolutions. Ces évolutions prennent en compte la définition de nouveaux modèles de transactions, de nouveaux protocoles de validation atomique et de nouveaux standards d'intégration. Tous ces éléments sont désormais isolés dans des services de transactions qui permettent aux développeurs de se concentrer sur les aspects métiers de leurs applications.*

Dans un second temps, nous étudions différents intergiciels de transactions académiques (services, canevas, etc.) qui ont adressé de différentes manières l'adaptabilité du domaine transactionnel. Pour ce faire, nous détaillons les différents critères d'évaluation que nous avons définis au regard des aspects que nous souhaitons adresser dans cette thèse. Cette évaluation est ensuite réalisée sur six solutions académiques proposées ces dernières années.

2.1 Introduction

Depuis son apparition dans les systèmes de gestion de base de données dans les années 70, la notion de transaction a pris un essor considérable dans la mesure où les transactions sont désormais employées à tous les niveaux applicatifs depuis les systèmes d'exploitation jusqu'aux applications de commerce électronique [GR93, PAB97, BCF+97]. Ces différentes applications se caractérisent par des systèmes d'informations complexes qui nécessitent un mécanisme fiable pour le maintien de la cohérence de leurs données.

La suite de ce chapitre est organisée comme suit. La section 2.2 introduit les notions essentielles du domaine transactionnel. La section 2.3 étudie différents travaux existants portant sur la prise en compte de l'adaptabilité dans les systèmes transactionnels. La section 2.4 établit une synthèse des approches existantes et positionne notre travail par rapport à ces approches. Enfin, la section 2.5 conclut ce chapitre.

2.2 Prérequis

Dans cette section, nous introduisons la notion de transaction, les propriétés ACID et les modèles de transactions existants. Puis nous présentons l'architecture usuelle des services de transactions et les différents standards transactionnels définis pour faciliter l'intégration des services de transactions.

2.2.1 Définition d'une transaction

Une transaction est l'exécution d'un programme contenant une séquence d'opérations (par exemple, des lectures et des écritures) réalisées sur des ressources (par exemple, bases de données, objets, composants, services). Les opérations sont supposées indivisibles, c.-à-d. que si deux opérations sont invoquées simultanément, l'une est exécutée totalement avant l'autre. L'exécution d'une transaction se termine soit par la validation soit par l'abandon. La validation d'une transaction rend définitif les effets de ses opérations sur les ressources tandis que son abandon les annule [BCF+97].

La figure 2.1 schématise les effets de l'exécution d'une transaction sur une donnée. La valeur de la donnée est représentée par l'axe des ordonnées tandis que le temps d'exécution est marqué par l'axe des abscisses. À l'initiation de la transaction, la donnée présente une valeur initiale et cohérente. Les opérations d'écriture successives sur cette donnée vont faire évoluer cette valeur dans le temps. La valeur à un instant donné de la transaction peut dépasser la limite de cohérence associée à cette donnée, c.-à-d. son invariant. Cependant, lors de la validation de la transaction, la valeur doit respecter l'invariant associé à la donnée afin d'être validée, dans le cas contraire la transaction doit être abandonnée et la donnée reprend sa valeur initiale.

Afin de contrôler les effets indésirables des modifications concurrentes des données d'un système, la transaction définit quatre propriétés identifiées sous l'acronyme **ACID** (atomicité, cohérence, isolation, durabilité) [GR93]. La mise en œuvre de ces propriétés requiert non seulement que l'application soit consciente de l'utilisation des transactions mais elle nécessite également un support dédié identifié sous le terme de **service de transactions**.

2.2.2 Propriétés ACID

La caractéristique d'une transaction réside dans le fait que l'exécution de toutes ses opérations doit laisser le système d'informations dans un état consistant. Pour ce faire, la transaction doit assurer quatre propriétés, appelées propriétés ACID, dont la définition communément admise est la suivante :

ATOMICITÉ : *une transaction est une unité complète et indivisible*. Soit une transaction se termine correctement et elle a les effets désirés sur les ressources manipulées (la transaction est

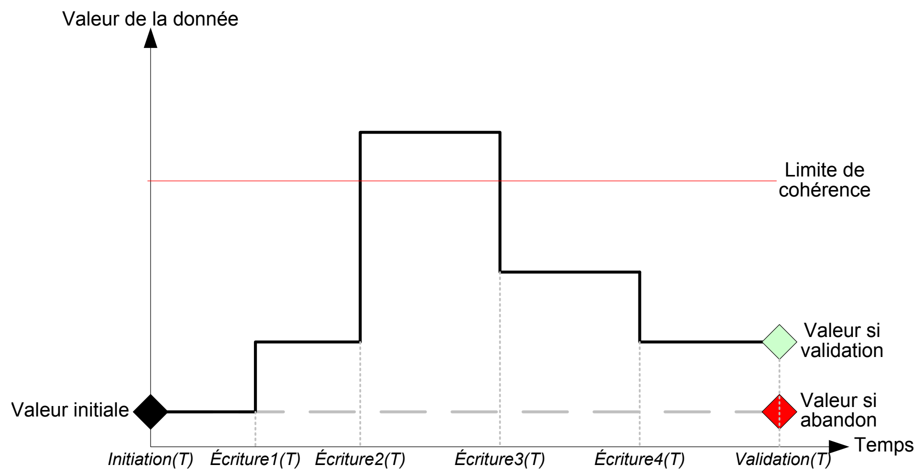


FIG. 2.1: Illustration de l'exécution d'une transaction.

validée), soit la transaction est interrompue et elle n'a aucun effet sur ces ressources (elle est abandonnée).

COHÉRENCE : *une transaction préserve la cohérence des ressources qu'elle manipule*. Les ressources manipulées par une transaction sont généralement logiquement reliés par des contraintes d'intégrité. Il en résulte que l'exécution d'une transaction doit préserver ces contraintes d'intégrité.

ISOLATION : *les effets d'une transaction préservent la cohérence des ressources qu'elle manipule*. Cette propriété vise à garantir à chaque transaction un accès aux ressources comme si elle était seule à s'exécuter dans le système.

DURABILITÉ : *les effets d'une transaction validée sont permanents et doivent survivre aux défaillances de la mémoire volatile du système*.

La garantie des propriétés ACID par un système d'informations quelconque assure que toute exécution concurrente d'un ensemble de transactions ne crée pas d'inconsistence dans les données de ce dernier.

La propriété de cohérence demeure de la responsabilité de l'application. Or, derrière les trois autres propriétés, il existe de nombreux protocoles de contrôle de concurrence pour assurer l'isolation des transactions, et des protocoles de reprise après défaillance afin de garantir les propriétés d'atomicité et de durabilité. Tous ces protocoles sont complexes et font l'objet d'une recherche active afin de fiabiliser la modification des données.

2.2.3 Modèles de transactions

Le modèle de transactions généralement associé aux propriétés ACID est le modèle de transactions plates [GR93]. Ce modèle est particulièrement adapté aux transactions s'exécutant en parallèle, de courtes durées et manipulant peu de données. Cependant, la diversité des contextes applicatifs actuels nécessite de définir de nouveaux modèles de transactions afin de supporter des transactions de longue durée (heures, jours, semaines), qui manipulent des données structurées et potentiellement volumineuses, et pour lesquelles un certain degré de coopération entre les utilisateurs peut être requis afin de réaliser une tâche complexe.

Limites des propriétés ACID. Les propriétés ACID doivent être relâchées car :

La propriété d'atomicité constitue une contrainte forte pour les transactions de longue durée. En effet, le risque d'abandon de la transaction croît de façon proportionnelle à sa durée. De

plus, le coût de l'abandon de la transaction croît également avec la durée et la complexité des traitements mis en œuvre.

La propriété d'isolation constitue un frein à l'exécution d'une transaction longue. En effet, la propriété d'isolation interdit toute externalisation des données modifiées avant la terminaison de la transaction modifiante (ce qui interdit toute coopération entre utilisateurs).

Modèles de transactions étendus. Il est donc nécessaire de définir de nouveaux modèles de transactions qui assouplissent les contraintes imposées par les propriétés transactionnelles ACID. Parmi les modèles de transactions étendus les plus connus, nous pouvons citer :

Le MODÈLE DE TRANSACTIONS EMBOÎTÉES permet de décomposer une transaction en une hiérarchie de sous-transactions pouvant s'exécuter en parallèle, chacune constituant une unité de reprise après défaillance [GR93]. Ce modèle pallie la rigidité du modèle de transactions plates en assouplissant la propriété d'atomicité et en contrôlant le parallélisme interne à chaque transaction. La figure 2.2 représente l'exécution d'une transaction emboîtée comportant trois niveaux. Toute transaction fille (par exemple, T12) initiée par une transaction mère (par exemple, T1) demeure sous sa responsabilité et doit se terminer avant cette dernière.

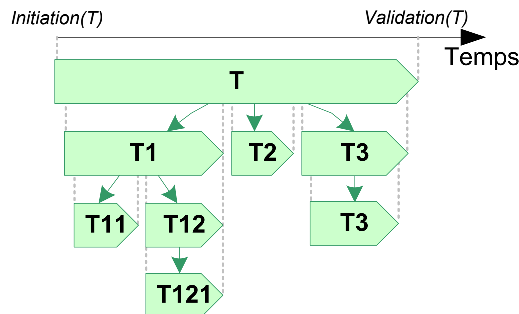


FIG. 2.2: Illustration du modèle de transactions emboîtées.

Le MODÈLE DE TRANSACTIONS MULTINIVEAUX peut être vu comme une spécialisation du modèle de transactions emboîtées permettant d'exploiter la commutabilité des opérations effectuées par les sous-transactions, afin d'accroître le degré de concurrence [BCF+97]. Il tire parti de la sémantique des ressources typées et de leur mode de construction hiérarchique pour offrir un contrôle de concurrence de plus haut niveau que celui réalisé par les transactions plates.

Le MODÈLE DE TRANSACTIONS SAGAS est adapté aux transactions de longue durée [GMS87]. Ce modèle relâche la contrainte d'isolation portant sur une transaction globale tout en garantissant la cohérence des resources de la base. Pour ce faire, le modèle SAGAS a recours à la définition de transactions de compensation pour annuler les effets d'une sous-transaction validée. La figure 2.3 présente un exemple d'exécution de transaction SAGAS. Dans ce modèle, la transaction T se contente de coordonner l'exécution des transactions qu'elle contient (les transactions T1, T2 et T3) et exécute les transactions de compensation nécessaires en cas d'abandon (les transactions C1, C2 et C3).

Le MODÈLE DE TRANSACTIONS COOPÉRANTES est beaucoup plus permissif et autorise des interactions complexes entre les utilisateurs [PAB97]. Ce modèle redéfinit les critères d'isolation entre un ensemble de transactions coopérantes. Le modèle de transactions *split/join* est un exemple de modèle de transactions coopérantes puisqu'il permet de répartir (opération *split*) et fusionner (opération *join*) dynamiquement les effets d'une transaction sur deux transactions indépendantes. Si ce modèle offre plus de flexiblité, il reporte néanmoins une partie importante du contrôle de cohérence sur l'application. La figure 2.4 illustre un

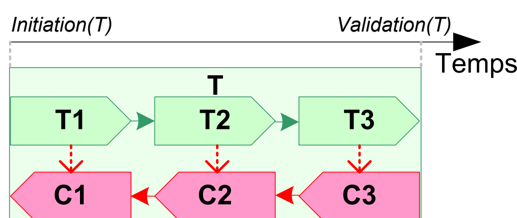


FIG. 2.3: Illustration du modèle de transactions SAGAS.

exemple de transaction utilisant le modèle *split/join*. La répartition d'une transaction T produit deux transactions T' et T" indépendantes qui peuvent éventuellement être fusionnées avec toute autre transaction supportant ce modèle.

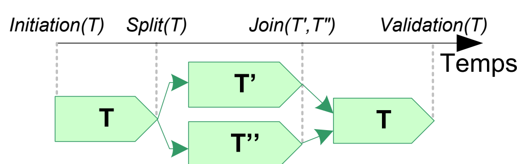


FIG. 2.4: Illustration du modèle de transactions coopérantes *split/join*.

Le MODÈLE DE TRANSACTIONS À FLOT DE TÂCHES permet de structurer des applications complexes accédant potentiellement à de nombreuses sources de données sous la forme d'un flot de tâches élémentaires, synchronisées et contrôlées [BCF+97]. Ce modèle consiste à contrôler un scénario d'exécution mettant en œuvre des traitements existants tout en conservant leur autonomie vis-à-vis des sources de données accédées. Ces traitements sont perçus comme des briques de base ACID auxquelles sont associées des actions de compensation afin de supporter l'abandon du scénario.

Formalisme ACTA. Pour faire face à la diversité des modèles de transactions étendus, le formalisme ACTA a été défini pour décrire le comportement et les propriétés de ces différents modèles [CR90, CR92]. Les règles ACTA permettent de caractériser les effets que les transactions produisent sur les autres transactions ainsi que sur les ressources qu'elles manipulent. Ces effets sont classifiés dans ACTA en utilisant les concepts de *visibilité*, de *conflit* et de *délégation* pour les effets associés aux ressources tandis que les effets associés aux transactions s'expriment en terme de *dépendances* entre les transactions.

2.2.4 Atomicité globale

La propriété d'atomicité assure que tout se passe comme si chaque transaction se terminait complétement ou ne s'exécutait pas. Dans un contexte centralisé, assurer la propriété d'atomicité nécessite un mécanisme de journalisation afin de refaire ou défaire les effets des transactions. Dans un contexte réparti, ce mécanisme de journalisation n'est pas suffisant car la répartition des participants sur plusieurs sites peut entraîner une divergence de décision sur l'issue des transactions. Dans ce cas, il est nécessaire d'assurer une propriété d'atomicité globale (plus connue sous le terme de *validation atomique répartie*). Pour ce faire, chaque participant exprime sa capacité à assurer les propriétés AID localement par un vote. Le vote vaut *oui* si le participant souhaite valider et vaut *non* dans le cas contraire. La décision globale de valider n'est prise que si tous les votes valent *oui*.

Propriété d'atomicité globale. De manière plus précise, le problème de la validation atomique répartie est définie par les quatre propriétés suivantes [BVVV05] :

UNANIMITÉ : *deux participants ne peuvent décider différemment*. Cette propriété reflète le consensus qui doit être accepté par tous les participants et assure l'atomicité globale de la transaction.

VALIDITÉ : *la transaction n'est validée que si tous les participants votent oui*. Cette propriété assure la durabilité et l'isolation de la transaction car les participants ne votent oui que s'ils sont en mesure de sauvegarder les effets de la transaction en mémoire permanente.

TERMINAISON : *si toutes les défaillances sont réparées et aucune défaillance ne survient, alors tous les participants doivent décider*. Cette propriété est une propriété de vivacité (par opposition aux précédentes propriétés qui sont des propriétés de sûreté). La terminaison impose qu'une fois les éventuelles défaillances réparées, une transaction doit valider ou abandonner.

NON-TRIVIALITÉ : *les participants doivent décider de valider si tous les votes sont oui et si aucun site n'est défaillant ou n'est suspecté de l'être*. Cette propriété évite les solutions triviales où les participants décident systématiquement d'abandonner toutes les transactions.

Protocole de validation en deux phases. Le protocole le plus communément utilisé pour résoudre le problème de la validation atomique répartie est le protocole de validation en deux phases (en anglais, *Two-Phase Commit - 2PC*) [Gra78]. Ce protocole fait l'objet d'une étude complète dans le chapitre 11 et par conséquent nous nous limitons ici à une présentation rapide du protocole.

Le site initiateur de la transaction joue le rôle de coordinateur comme illustré dans la figure 2.5. Le protocole est appelé validation à deux phases à cause des deux étapes qui le composent. Durant la phase de vote, le coordinateur demande à tous les participants leur vote par l'envoi d'une message *prepare*. Tous les participants envoient leur vote au coordinateur. Durant la phase de décision, le coordinateur décide en fonction des votes reçus de valider la transaction si tous les votes sont *oui* et d'abandonner dans le cas contraire. La décision est transmise aux participants par l'envoi d'un message *commit* (dans le cas d'une validation) ou *abort* (dans le cas d'un abandon). Les participants appliquent localement la décision du coordinateur et envoyent un acquittement *acknowledge* au coordinateur. Lorsque le coordinateur a reçu les acquittements de tous les participants, la transaction est considérée comme terminée.

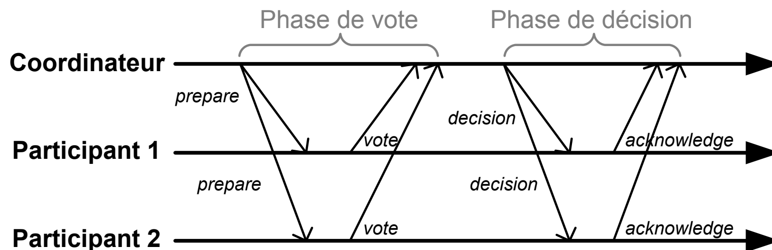


FIG. 2.5: Représentation du protocole de validation en 2 phases.

Chaque phase fait l'objet de journalisations de la part du coordinateur et de chaque participant afin de pouvoir redémarrer après une éventuelle défaillance. Si une défaillance survient, le participant exécute une procédure de reprise pour retrouver la décision et terminer la transaction. Cette procédure est différente selon qu'il s'agit d'une reprise du coordinateur ou d'un participant.

Optimisations des protocoles de validation. Les performances d'un protocole de validation en deux phases impactent le temps d'exécution de la transaction et le temps d'occupation des ressources réservées par les participants. Ces performances affectent également le temps d'attente des autres transactions du système. Dès lors, différentes optimisations ont été définies afin d'améliorer les performances des protocoles de validation dans des contextes d'exécution particuliers. Le chapitre 11 étudie et optimise le coût des échanges de messages et de la journalisa-

tion de différentes optimisations du protocole de validation en deux phases, cependant il existe d'autres optimisations de ce protocole.

Les *transactions de lecture* permettent de ne pas impliquer dans le protocole de validation les participants n'opérant que des lectures. De plus, si un seul participant opérant des écritures participe au protocole de validation, alors la validation en deux phases est inutile.

Le *protocole de validation en deux phases décentralisé* réduit le nombre d'étapes de communication en permettant aux participants votant *non* d'indiquer directement leur décision aux autres participants (voir figure 2.6). Sa généralisation permet à tous les participants de diffuser leur décision aux autres participants afin que ceux-ci puissent inférer la décision globale localement à partir des votes reçus par chaque participant.

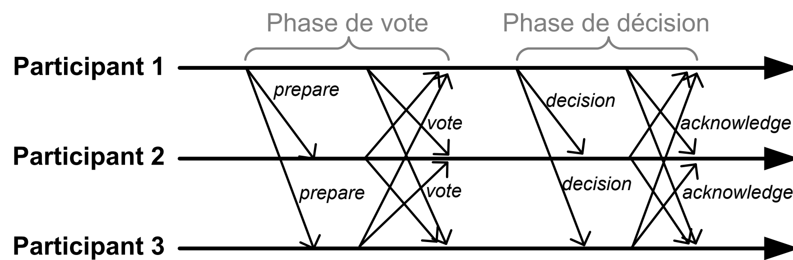


FIG. 2.6: Représentation du protocole de validation en 2 phases décentralisé.

Le *protocole de validation en deux phases hiérarchique* permet de considérer des topologies de réseaux dans lesquelles tous les sites ne sont pas directement connectés. Dans ce cas, certains participants peuvent jouer le rôle d'intermédiaire pour le protocole de validation en relayant et en interprétant localement les décisions des participants qu'il contacte (comme illustré dans la figure 2.7).

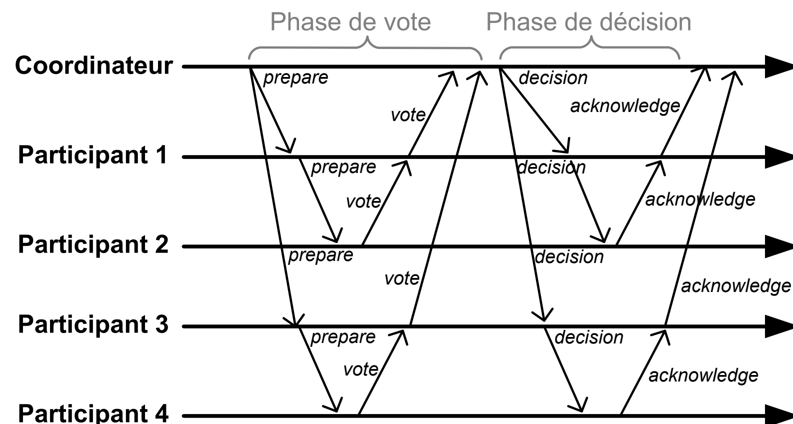


FIG. 2.7: Représentation du protocole de validation en 2 phases hiérarchique.

Le *protocole de validation en trois phases* (3PC) offre une solution au problème de blocage du protocole 2PC qui peut survenir lors de la défaillance du coordinateur. Comme illustré dans la figure 2.8, le protocole est bloquant lorsque le coordinateur tombe en panne car les participants ne peuvent terminer le processus de validation tant qu'ils n'ont pas reçu la décision finale à appliquer.

Pour résoudre ce problème, le protocole 3PC ajoute une phase d'intention de décision au protocole 2PC et il s'assure que cette intention a bien été reçue par tous les participants avant de communiquer sa décision définitive (voir figure 2.9). L'ajout de cette phase permet aux participants de se concerter pour récupérer la décision finale quel que soit le moment où survient la défaillance du coordinateur.

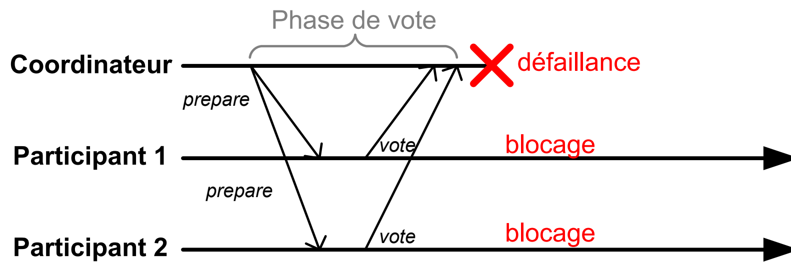


FIG. 2.8: Blocage dans l'exécution du protocole de validation en 2 phases.

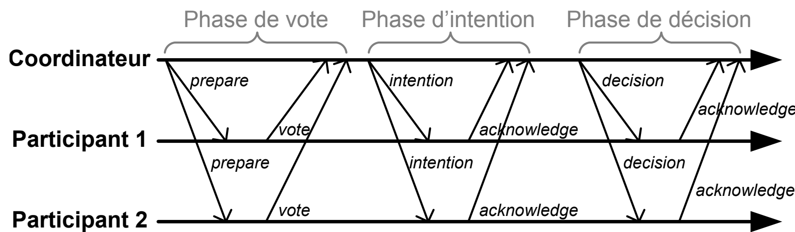


FIG. 2.9: Représentation du protocole de validation en 3 phases.

2.2.5 Services de transactions

Si les principes fondateurs du concept de transaction sont issus du domaine des bases de données, l'évolution des systèmes centralisés vers les systèmes répartis et plus récemment vers les systèmes répartis à objets/composants/services a entraîné une généralisation de l'utilisation du concept de transaction. Pour simplifier la tâche des développeurs de systèmes d'informations, les protocoles transactionnels sont désormais isolés dans des solutions dédiées plus connues sous le nom de *services de transactions* (également appelé moniteur de transactions dans les bases de données). Ces services de transactions se chargent de fournir les propriétés d'atomicité, d'isolation et de durabilité, et laissent le développeur du système d'informations se concentrer sur les aspects de cohérence.

Dès lors, ces services de transactions peuvent prendre la forme d'un objet dédié disposant d'interfaces de programmation identifiées permettant à toute application de décrire des séquences opératoires devant s'exécuter dans un contexte transactionnel. Dans la figure 2.10, l'application contacte le service de transactions pour débuter une transaction (a) puis exécute une séquence d'opérations métiers (b). Chaque opération modifiant une donnée du système a pour effet l'acquisition du verrou associé à la donnée par la transaction en cours d'exécution (c) et l'enregistrement de la source de données associée (base de données, fichier, etc.) comme un participant de la transaction (d). À l'issue de la séquence d'opérations, l'application ordonne la validation de la transaction (e). Cet ordre déclenche l'exécution d'un protocole de validation implanté par le service de transactions entre toutes les sources de données concernées par l'exécution de la transaction (f).

Le contrôle de concurrence peut également être isolé sous la forme d'un *service de concurrence* afin de simplifier la gestion des sections critiques associées à l'exécution des transactions. De ce fait, le développeur peut se concentrer sur les préoccupations métiers de l'application et déléguer la réalisation des préoccupations techniques à un ensemble de services dédiés. L'identification de ces services facilite également la synchronisation de données situées sur des sites distants en offrant des services globaux accessibles à distance. C'est ainsi que la popularité grandissante de ces services a donné lieu à de nombreux efforts de standardisation de la fonctionnalité transactionnelle.

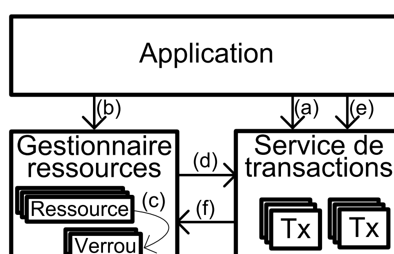


FIG. 2.10: Représentation d'un service de transactions.

2.2.6 Standards transactionnels

Il est de plus en plus fréquent de construire les systèmes transactionnels de façon modulaire sous forme de composants. Néanmoins, l'intégration de ces composants de sources hétérogènes nécessite la définition d'interfaces communes afin de permettre à ces derniers de pouvoir communiquer. Les standards transactionnels constituent une réponse à cette préoccupation.

Standard DTP. Le consortium X/Open a défini un ensemble d'interfaces pour composants transactionnels dans le cadre de son architecture DTP (en anglais, *Distributed Transaction Processing*) [The96]. Les interfaces de ce standard les plus connues sont les interfaces XA et TX [The92]. L'interface XA permet de coordonner la validation d'une transaction accédant à plusieurs bases de données à partir d'un coordinateur global. L'interface TX permet de contrôler la transaction depuis l'application.

La figure 2.11 présente ces interfaces et leur association avec les composants transactionnels *Application*, *Gestionnaires de ressources* et *Service de transactions*. Le terme *interfaces natives* fait référence à un jeu d'interfaces arbitrairement définies entre l'application et le gestionnaire de ressources, c.-à-d. non standardisées par DTP.

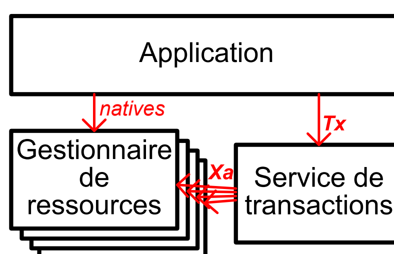


FIG. 2.11: Représentation du standard DTP.

Standard OTS. Le consortium *Object Management Group* (OMG) a standardisé la fonctionnalité transactionnelle dans le cadre de son architecture CORBA (*Common Object Request Architecture*) [OMG04]. Ainsi, le service OTS (*Object Transaction Service*) définit non seulement des interfaces de programmation *CosTransactions* en utilisant le langage de définition d'interfaces (*Interface Definition Language* [IDL]), mais aussi quatre modèles de programmation [OMG03]. Ces quatre modèles de programmation correspondent aux combinaisons des modes d'utilisation *directe* ou *indirecte*, et des modes de propagation du contexte transactionnel *explicite* ou *implicite*.

La figure 2.12 résume les fonctionnalités du standard OTS. Dans le mode de programmation directe, l'application manipule les interfaces OTS tandis que dans le mode de programmation indirecte, elle utilise un objet intermédiaire *Current* qui se charge d'accéder aux interfaces OTS. Dans le mode de propagation explicite, l'application transmet le contexte transactionnel par les paramètres des opérations invoquées. Le mode de propagation implicite transmet le contexte

transactionnel en utilisant le mécanisme d'intercepteurs portables mis à disposition par l'architecture CORBA [WPSO01]. Enfin, le standard OTS supporte aussi bien les transactions plates que les transactions emboîtées.

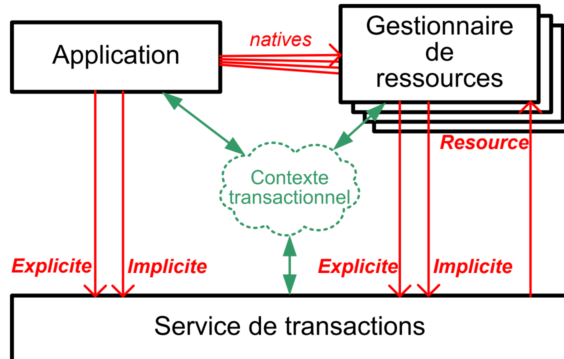


FIG. 2.12: Représentation du standard OTS.

Standard JTS. Le groupe de standardisation *Java Community Group* (JCP) a produit deux spécifications pour offrir un support transactionnel aux applications Java. Ainsi, la spécification JTA (*Java Transaction API*) définit les interfaces de programmation accessibles par une application Java. Cette spécification supporte les interfaces définies par le standard DTP comme standard d'intégration des sources de données. La spécification JTS (*Java Transaction Service*) présente l'architecture d'un service de transactions fournissant les interfaces JTA [Che99]. Cette architecture repose sur le standard OTS afin de propager le contexte transactionnel entre les services de transactions de type JTS. La spécification JTS applique exclusivement un modèle de transactions plates.

La figure 2.13 présente l'architecture du standard JTS. Cette architecture repose sur l'architecture 3-tiers des serveurs d'applications J2EE en offrant différentes interfaces aux trois niveaux de l'application. Ainsi, l'interface *UserTransaction* est manipulée par l'application alors que le serveur d'applications utilise l'interface *TransactionManager*. Les sources de données manipulées par l'application peuvent enregistrer des ressources dans la transaction via l'interface *Xa* de la spécification JTA. L'implémentation du service de transactions repose ensuite sur un gestionnaire de communication de type IIOP (en anglais, *Internet Inter-ORB Protocol*) pour propager le contexte transactionnel entre services de transactions JTS (ou OTS) compatibles.

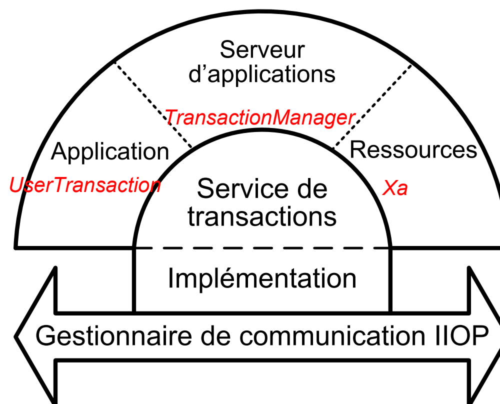


FIG. 2.13: Représentation du standard JTS.

Standards WS-AT & WS-BA. Récemment, le groupe de standardisation des *Web Services* a défini de nouvelles spécifications pour la prise en compte de la gestion des transactions lors des interactions entre Web Services. Ainsi, la spécification WS-AT (*Web Services - Atomic Transactions*) [CCF+05a] propose un service de transactions supportant le modèle de transactions plates tandis que la spécification WS-BA (*Web Services - Business Activities*) [CCF+05c] prend en charge le modèle de transactions à flot de tâches.

La figure 2.14 présente l'architecture des services de transactions du type WS-AT. Cette architecture fait intervenir la spécification WS-Coord (*Web Services - Coordination*) [CCF+05b] sous la forme d'un service de coordination pour le support du protocole de validation en deux phases. Le contexte transactionnel (TX CTX) est propagé entre les différentes éléments applicatifs via l'entête des requêtes SOAP (en anglais, *Simple Object Access Protocol*) émises par l'application ou les différents services. Le service WS-Coord présente des interfaces décrites avec le langage WSDL (*Web Services Description Language*) [CMRW06] afin de définir un groupe de participants (Activation) et d'enregistrer des participants (Registration). Le service WS-AT fournit des interfaces pour contrôler l'exécution de la transaction (Completion) et du protocole de validation (2PC).

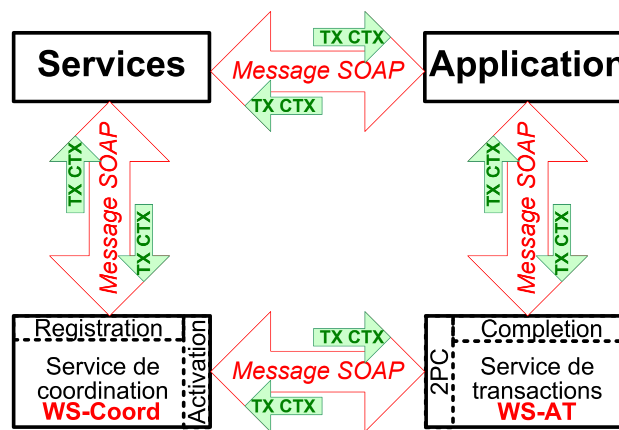


FIG. 2.14: Représentation du standard WS-AT.

2.2.7 Synthèse

Cette section a présenté succintement les principes de la gestion des transactions dans les intergiciels et les différentes stratégies transactionnelles disponibles. Ces stratégies comportent non seulement de nombreux modèles de transactions et protocoles de validation atomique, mais elles sont également régies par différents standards transactionnels qui ne cessent d'évoluer (de manière concurrente ou non). Cependant, ces standards présentent de nombreuses similarités qui s'expliquent par le fait que la plupart des nouveaux standards intègrent les fonctionnalités des standards existants en se contentant de les englober dans une nouvelle couche. Une étude approfondie des fonctionnalités des standards OTS, JTS et WS-AT est présentée dans le chapitre 10.

Pour faire face à l'explosion combinatoire des différentes combinaisons de stratégies et de standards réalisables, de nombreux travaux académiques ont proposé des architectures de services extensibles pour prendre en compte différentes stratégies.

2.3 Étude de l'existant

Dans cette section, nous étudions différents canevas logiciels qui ont été définis ces dernières années pour résoudre les limitations des approches transactionnelles classiques. Ainsi, nous

présentons le canevas logiciel *Reflective Transaction Framework* proposant une extension modulaire des services de transactions existants pour supporter les modèles de transactions étendus. Ensuite, nous étudions le canevas logiciel *Kernel Aspect Language for ATMS* (KALA) dont l'objectif est d'isoler et de simplifier l'intégration du code de démarcation transactionnelle pour les modèles de transactions étendus. Nous abordons également les travaux réalisés autour de l'adaptation dynamique des services de transactions en étudiant les approches définies dans REFLECTS et le service d'adaptation des services techniques. Nous étudions également le service de transactions ARJUNATS pour sa capacité à s'adapter à différents standards transactionnels. Enfin, nous observons d'autres approches relatives au domaine transactionnel telle que l'approche BOURGOGNETS ou le modèle *Open Nested Transactions*.

2.3.1 Critères d'évaluation

Notre étude des approches existantes dans le domaine des intergiciels transactionnels repose sur six critères que nous jugeons comme pertinents pour l'évaluation des intergiciels transactionnels adaptables.

OUVERTURE : l'ouverture est la capacité d'un intergiciel à supporter des adaptations *a posteriori* non anticipées lors du développement de l'intergiciel. Cette propriété assure la pérennité de l'intergiciel dans le temps face à l'évolution du contexte dans lequel il est intégré. Les questions que nous nous posons lors de l'évaluation de la propriété d'ouverture d'un intergiciel sont : *l'intergiciel fournit-il un ou plusieurs axes d'adaptabilité ou d'extension ? Quel type d'adaptation est supporté ? Quels sont ses avantages et inconvénients ?*

GRANULARITÉ : la granularité correspond aux paradigmes de la technologie logicielle ou du modèle de programmation utilisé pour développer l'intergiciel. La granularité de l'intergiciel peut impacter non seulement le degré d'ouverture de celui-ci mais aussi ses performances. L'évaluation de la propriété de granularité d'un intergiciel soulève les questions suivantes : *l'intergiciel utilise-t-il une technologie particulière ? Quels sont les paradigmes exploités ?*

MODULARITÉ : la modularité représente le fait de pouvoir appliquer des modifications à une partie du système sans en affecter les autres parties. Cette propriété généraliste permet d'évaluer le support de la complexité par l'intergiciel et sa faculté à isoler les composantes caractéristiques de ses fonctionnalités. La modularité ne repose pas uniquement sur le choix d'une granularité adaptée et d'une technologie particulière mais dépend surtout d'une réflexion sur l'identification des points d'interaction entre les modules afin de minimiser l'impact de la reconfiguration d'un module sur les modules qui lui sont liés. Lors de l'évaluation de la propriété de modularité d'un intergiciel, nous observons les points suivants : *l'intergiciel est-il défini de manière modulaire ? Quelle structure est proposée pour faciliter cette modularité ?*

CONFIGURATION : la configuration d'un intergiciel correspond au formalisme proposé à l'utilisateur pour adapter l'intergiciel à ses besoins applicatifs. Un intergiciel configurable doit fournir un formalisme proche de l'utilisateur facilitant son intégration dans l'environnement d'exécution. Les questions que nous nous posons lors de l'évaluation de la propriété de configuration d'un intergiciel sont : *l'intergiciel présenté est-il et/ou peut-il être configuré ? Quel formalisme est proposé pour décrire la configuration ?*

INTÉGRATION : l'intégration d'un intergiciel adresse sa capacité à être utilisé dans un contexte applicatif sans remettre en cause la plate-forme dans laquelle il est intégré. Cette capacité consiste à faciliter l'intégration de l'intergiciel dans le système en se basant sur des standards d'intégration reconnus ou en utilisant des mécanismes d'intégration transparents pour l'intégrateur. Les questions que nous nous posons lors de l'évaluation de la propriété d'intégration d'un intergiciel

sont : *l'intergiciel présenté peut-il s'intégrer dans une plate-forme applicative ? Se base-t-il sur un standard reconnu pour faciliter son intégration ? Cette intégration est-elle transparente ?*

PERFORMANCE : la performance liée à l'exécution d'un intergiciel transactionnel est cruciale. En effet, les intergiciels transactionnels prennent en charge une partie de la tolérance aux fautes d'une application et leur utilisation doit minimiser le surcoût à l'exécution de l'application tout en maximisant la fiabilité de l'application. La question que nous nous posons lors de l'évaluation de la propriété de performance d'un intergiciel est : *l'intergiciel présenté introduit-il un surcoût à l'exécution comparé aux approches traditionnelles ?*

Dans la suite de cette section, nous présentons différents travaux menés sur les intergiciels transactionnels et nous appliquons nos critères d'évaluation sur ces derniers.

2.3.2 *Reflective Transaction Framework*

Présentation. La définition du canevas ACTA [CR90, CR94] a facilité la formalisation de nombreux modèles de transactions étendus (transactions emboîtées, SAGAS, coopérantes, etc.). Cependant, le nombre d'implémentations de ces modèles de transactions étendus demeure relativement limité de par leur complexité de mise en œuvre et de par la disparité des services de transactions atomiques disponibles. De plus, il existe très peu d'implémentations supportant plus d'un modèle de transactions étendu.

Le canevas logiciel *Reflective Transaction Framework* a été défini afin de répondre à ces limitations [BP97, Bar98]. L'objectif de ce canevas logiciel est de pouvoir étendre les services de transactions atomiques existants avec un mécanisme suffisamment flexible pour mettre en œuvre différents modèles de transactions étendus. Pour ce faire, ce canevas logiciel combine les bénéfices de travaux réalisés sur l'approche ASSET (*A System for Supporting Extended Transactions*) [BDG+94] et TSME (*Transaction Specification and Management Environment*) [GHKM94]. En effet, chacune de ces approches fournit une syntaxe dédiée pour décrire le comportement des modèles de transactions étendus. C'est ainsi que le canevas *Reflective Transaction Framework* définit un langage extensible de description de la sémantique des modèles de transactions étendus.

L'approche définie dans le canevas *Reflective Transaction Framework* propose d'étendre les fonctionnalités d'un service de transactions existant avec celles des modèles de transactions étendus [BP95]. Pour ce faire, le canevas favorise la séparation des interfaces du service de transactions et définit un protocole à méta-objet (*Meta-Object Protocol* [MOP]) afin de fournir une implantation modulaire des modèles de transactions étendus. La figure 2.15 résume l'architecture de ce canevas. Le canevas identifie trois niveaux d'interfaces : Transaction Demarcation Interface, Extended Transaction Interface et Meta-Interface. Le niveau Transaction Demarcation Interface définit les opérations d'extension de la sémantique des transactions ACID. Le niveau Extended Transaction Interface définit les opérations propres au modèle de transactions étendu implanté par le canevas. Enfin, le niveau Meta-Interface étend le service de transactions existant et fournit les opérations nécessaires à la définition de la sémantique des deux premiers niveaux.

Ces trois niveaux sont délivrés par un adaptateur de service de transactions (**Transaction Manager Adapter**). Cet adaptateur est un module additionnel construit au dessus d'un service de transactions existant (**Legacy Transaction Monitor**) pour étendre ses fonctionnalités. Pour ce faire, celui-ci définit un protocole à méta-objet basé sur la notion de Meta-transaction [BP96]. Ce méta-objet est associé à un descripteur qui associe les fonctionnalités du service de transactions étendu à leur sémantique (Implementation Level).

A la réception d'une invocation d'une opération du service de transactions **(1)**, le protocole à méta-objet exécute la méta-transaction associée à la transaction de base (Base Level Transaction) **(2)**. La méta-transaction consulte son méta-descripteur pour connaître la sémantique associée à cette opération **(3)**. Elle délègue ensuite l'exécution de la sémantique à un objet d'implémentation

qui utilise les trois niveaux d'interfaces pour décrire le comportement que la transaction doit appliquer **(4)** avant de laisser le contrôle à la transaction de base **(5)**. Les invocations sur l'objet d'implémentation peuvent être contrôlées par deux invariants (*preTest Invariant* et *postTest Invariant*).

FIG. 2.15: Architecture de *Reflective Transaction Framework*.

Synthèse. Le canevas *Reflective Transaction Framework* adresse l'**adaptation des modèles de transactions** fournis par des services de transactions existants. La **modularité** de l'architecture de ce canevas repose sur l'utilisation d'un *protocole à méta-objet* dédié. Ce protocole permet d'adapter le comportement du service de transactions à la **granularité d'une méthode**. La **configuration** du comportement est réalisée via un *méta-descripteur* qui associe les opérations fournies par le service de transactions à leur sémantique étendue. Cette architecture offre une structure extensible permettant au développeur de définir de nombreux modèles de transactions étendus (par exemple, modèle de transactions *split/join*, coopératives). Cependant, le canevas n'offre aucun support pour modifier le protocole de validation utilisé par le service de transactions. Les **performances** de ce service de transactions ne sont pas abordées dans la littérature mais l'utilisation d'un protocole à méta-objet pour étendre le comportement d'un service de transactions laisse présager une *dégradation des performances* liée à l'utilisation du canevas. Enfin, l'**intégration** du canevas repose sur la définition de l'interface Extended Transaction Interface. Cette interface spécifie les opérations spécifiques au modèle de transactions étendu supporté par le canevas. Cependant, le contenu de cette interface ne repose sur *aucun standard transactionnel* reconnu.

2.3.3 *Kernel Aspect Language for ATMS*

Présentation. L'approche présentée dans KALA (*Kernel Aspect Language for ATMS*) [Fab05] adresse le problème de l'utilisation des modèles de transactions étendus (*Advanced Transaction Mechanisms* [ATMS]). En effet, si la définition des ATMS permet de répondre aux besoins de flexibilité des applications, leur utilisation demeure très complexe pour le développeur et impose à ce

dernier de mélanger le code fonctionnel avec le code de démarcation de l'ATMS. Dans certaines situations, le code technique lié à la démarcation de l'ATMS peut même représenter un volume plus important que le code métier de l'application.

Pour faire face à cette limitation des ATMS, l'approche introduite dans KALA propose d'appliquer les principes de la *programmation par aspects* [KLM+97] pour modulariser la gestion de la démarcation transactionnelle au niveau de l'application. L'objectif est d'extraire le code relatif à la démarcation de l'application afin de pouvoir le modulariser et l'injecter automatiquement en utilisant un mécanisme de tissage (**Aspect Weaver** dans la figure 2.16). Cependant, la démarcation transactionnelle de l'ATMS intègre de nombreuses préoccupations qui compliquent l'extraction du code (délégation des verrous, gestion des dépendances, gestion des conflits, gestion des abandons, etc.). Cette difficulté est résolue par l'identification d'un squelette d'intégration de la démarcation transactionnelle (**Skeleton** dans la figure 2.16) comportant cinq étapes. L'étape `Preliminaries` identifie la phase d'initialisation nécessaire au bon fonctionnement du code de démarcation. L'étape `Beginning` identifie le code de démarcation qui doit être réalisé avant l'exécution du code métier de l'application. L'étape `Running` correspond au code métier original de l'application. L'étape `Commit` identifie le code de démarcation qui doit être intégré pour gérer la validation d'une transaction. Enfin, l'étape `Abort` identifie le code démarcation qui doit être intégré pour gérer l'abandon d'une transaction. Ce squelette d'intégration de la démarcation est indépendant du type d'ATMS utilisé par le développeur de l'application. Ce squelette est utilisé par le tisseur dédié à la démarcation transactionnelle (**Translator** dans la figure 2.16) pour intégrer le code technique dans l'application.

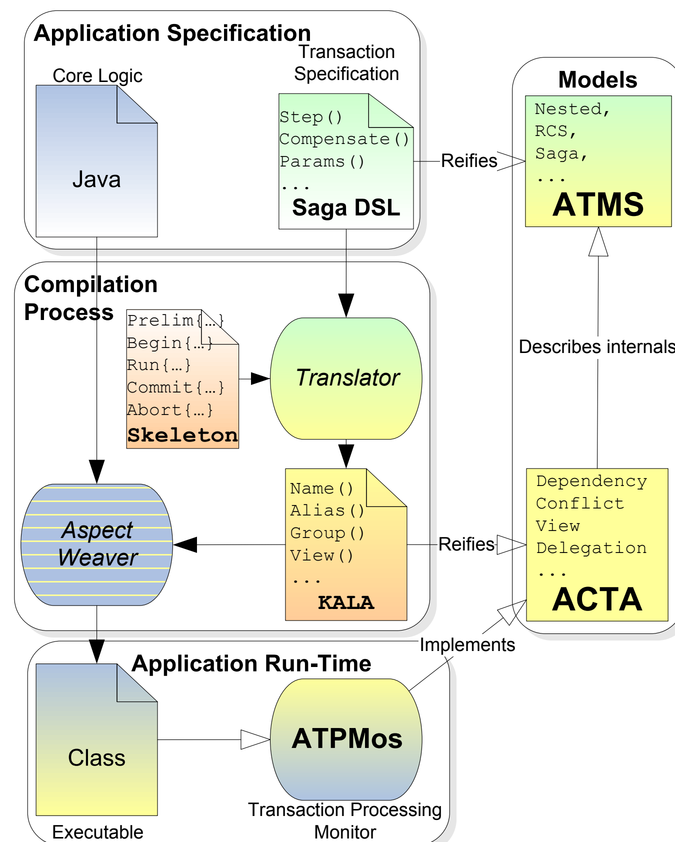


FIG. 2.16: Architecture de l'approche KALA.

Afin de décrire le code de démarcation transactionnelle à intégrer au niveau de chaque étape du squelette d'intégration, KALA utilise une réification du canevas ACTA [CR94] sous

la forme d'un langage dédié dont les primitives correspondent aux concepts identifiés dans le canevas ACTA (`Name()`, `Alias()`, `Group()`, `View()`, etc.) [FD06b]. Pour chacune des étapes d'intégration, KALA utilise ce langage pour déclarer les opérations de démarcation à effectuer selon l'ATMS considéré. Dès lors, le code technique de démarcation d'un ATMS est isolé du code métier lié à l'application. Cependant, la manipulation du langage KALA demeure difficile dans la mesure où les primitives proches du formalisme ACTA ne sont pas toujours intuitives pour les développeurs de l'application. De plus, ces primitives doivent être utilisées à bon escient afin de respecter le comportement d'un ATMS donné.

Pour limiter les erreurs de démarcation transactionnelle, KALA propose de définir un ensemble de langages dédiés à la démarcation transactionnelle des ATMS [FC05, FD06a]. L'objectif est de fournir un langage dédié à la démarcation pour chaque ATMS. Chaque langage définit des primitives de plus haut niveau que celles disponibles dans le langage KALA. La figure 2.16 illustre l'utilisation d'un langage dédié à l'ATMS SAGAS (**SAGAS DSL** dans la figure 2.16) et définissant des primitives `Step()`, `Compensate()` et `Params()` pour décrire le comportement de démarcation transactionnelle à appliquer pour le modèle de transactions étendu SAGAS [GMS87]. La sémantique des primitives de ce langage est définie à partir des opérateurs définis dans le canevas ACTA. Cette définition utilisant le DSL SAGAS est ensuite convertie vers une description équivalente utilisant le langage KALA.

La définition KALA de démarcation des transactions SAGAS est ensuite tissée sur le code de l'application. Dès lors, l'application supporte la démarcation pour l'ATMS SAGAS et utilise le service de transactions **ATPMos** (*ATMS Transaction Processing Monitor*) pour réaliser le comportement transactionnel. Le service de transactions **ATPMos** fournit une implantation de la réification des concepts du canevas ACTA utilisés dans le langage KALA.

Synthèse. Le canevas logiciel KALA adresse l'**intégration** de la démarcation pour les modèles de transactions étendus (ATMS). Pour ce faire, KALA exploite les principes de la *programmation par aspects* afin d'isoler et de modulariser le code technique lié à la démarcation transactionnelle. La **configuration** de cette intégration est réalisée en utilisant l'un des différents *langages dédiés* à la démarcation des ATMS supportés par KALA. Chacun de ces langages définit des primitives de haut niveau qui sont spécifiques au modèle de transactions étendu considéré, et qui peuvent être converties en une séquence de primitives de plus bas niveau inspirée du canevas ACTA. D'ailleurs, l'**adaptabilité** de KALA repose sur la richesse de son langage généraliste réifiant les concepts du canevas ACTA. La **granularité** d'adaptation dépend des primitives de bas niveau réifiées à partir du canevas ACTA. Les constructions de ce langage sont dirigées par un *squelette d'intégration* identifiant les cinq étapes spécifiques à la démarcation pour les ATMS Ainsi, la **modularité** supportée par KALA repose sur le squelette d'intégration qui est utilisé par le tisseur d'aspects pour intégrer le code technique de la démarcation transactionnelle. Enfin, KALA n'introduit pas de surcoût à l'exécution car le code technique tissé par cette approche est équivalent au code que le développeur doit écrire en temps normal. Les **performances** résultantes sont par conséquent dépendantes du service de transactions sous-jacent utilisé par KALA : ATPMos.

2.3.4 REFLECTS

Présentation. Le canevas logiciel REFLECTS a été défini pour prendre en compte l'évolution des contextes d'exécution des applications. Cette évolution requiert non seulement de nouvelles propriétés transactionnelles mais aussi un support d'exécution dédié pour faire face à la dynamicité des variations. Ainsi, une même application peut requérir à différentes étapes de son cycle de vie des modèles de transactions différents pour répondre à la variation des besoins de tolérance aux fautes et de contrôle de concurrence de l'application.

Pour répondre à ces besoins de dynamicité, le canevas REFLECTS propose un mécanisme de courtage étendu pour adapter le service de transactions aux besoins de l'application [KJ03, Kar03]. Pour ce faire, REFLECTS propose de caractériser les propriétés des services de transactions existants et fournit à l'application un mécanisme de sélection du service de transactions

adapté à ses besoins [AK05]. L'application exprime alors les propriétés transactionnelles qu'elle requiert et REFLECTS fournit l'instance de service de transactions adapté à ses besoins. REFLECTS est construit en appliquant les principes de la conception par composants [SGM02] et utilise le modèle de composants OPENCOM [CBG+04] (présenté dans le chapitre suivant en section 3.3.2).

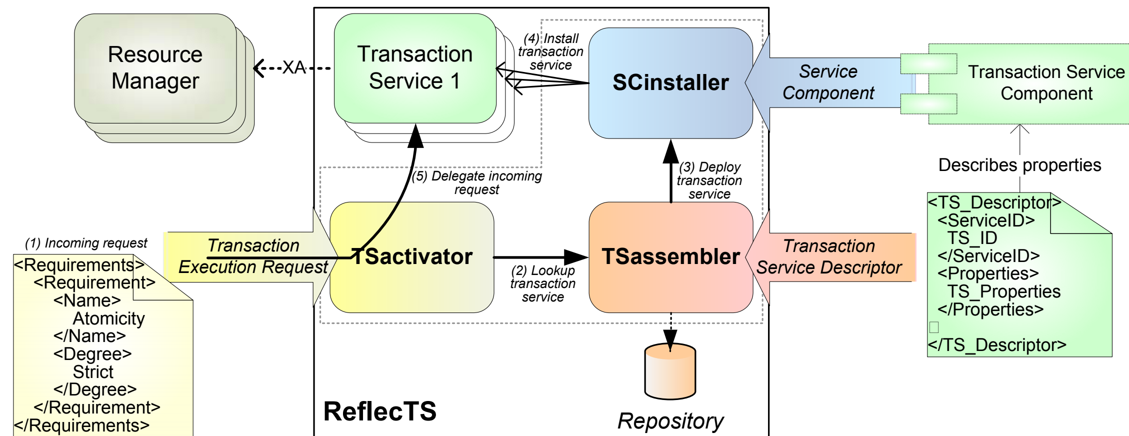


FIG. 2.17: Architecture du canevas REFLECTS.

La figure 2.17 résume l'architecture du canevas REFLECTS. Le courtier se compose de trois composants : `TSactivator`, `TSassembler` et `SCinstaller`. Le composant `SCinstaller` permet d'enregistrer une archive de service de transactions existant décrite sous la forme d'un composant déployable. Le composant `TSassembler` permet d'enregistrer une description des propriétés d'un service de transactions existant et de l'associer à son archive préalablement enregistrée auprès du composant `SCinstaller`. Lorsque le composant `TSactivator` reçoit une requête d'exécution d'une transaction provenant de l'application *(1)*, le composant utilise la description des besoins transactionnels de l'application associée à la requête pour interroger le composant `TSassembler` *(2)*. Cette description des besoins est basée sur un ensemble de critères extensibles (`Atomicity`, `Consistency`, `Isolation`, `Durability`, `Time`, etc.) auxquels sont associés différentes valeurs (`Strict`, `Relaxed`, `Alternative`, `Exception`, etc.). Le composant `TSassembler` interroge alors le référentiel des descripteurs de services de transactions disponibles (*Repository*). Le processus de sélection peut trouver plusieurs candidats susceptibles pour une description donnée des besoins de l'application. Dans ce cas, ce sont les services de transactions déjà disponibles qui sont favorisés par le processus de sélection. Par contre, si le service de transactions associé au descripteur sélectionné par le composant `TSassembler` n'est pas encore installé, le composant `TSassembler` demande au composant `SCinstaller` de procéder à l'installation du service de transactions dans l'environnement REFLECTS *(4)*. Une fois le service de transactions installé, le composant `TSactivator` récupère le contrôle et oriente la requête de l'application vers le service de transactions sélectionné.

Synthèse. REFLECTS adresse l'**adaptabilité** dynamique des services de transactions utilisés par les applications requiérant l'usage de différents modèles de transactions au cours de leur exécution. Par conséquent, la **granularité** d'adaptation supportée par REFLECTS se situe au niveau du service de transactions. En effet, en se basant sur un *mécanisme de courtage étendu*, REFLECTS est capable d'orienter les requêtes transactionnelles d'une application vers l'instance de service de transactions disponible la plus adaptée. La mise en œuvre de REFLECTS applique les principes de la *programmation par composants* et utilise en particulier le modèle de composants OPENCOM. La **modularité** de REFLECTS repose ainsi sur une architecture constituée de trois composants. Les composants `CSinstaller` et `TSassembler` assurent la **configuration** de l'intergiciel du service de courtage. Le composant `SCinstaller` supporte l'ajout dynamique de nouveaux services de transactions et se charge de les installer au sein de l'environnement

d'exécution de REFLECTS. Le composant `TSinstaller` maintient la liste des services de transactions disponibles et leurs caractéristiques. Le composant `TSactivator` assure l'opération de sélection du service de transactions adapté aux besoins de l'application. Les **performances** de REFLECTS reposent donc principalement sur cette dernière fonction de sélection. Dès lors, un surcoût est introduit lors de chaque accès au service de transactions à cause du mécanisme de routage imposé par REFLECTS. De plus, la granularité d'adaptation limite les possibilité d'adaptation de l'approche. Enfin, l'**intégration** de REFLECTS requiert que l'application puisse exprimer dynamiquement ses besoins transactionnels.

2.3.5 Service d'adaptation des services techniques

Présentation. Les travaux abordés dans [HL04, Hér05] étendent les principes définis dans le cadre du canevas intergiciel REFLECTS. Cette extension consiste non seulement à élargir le champ d'application du service de courtage en le généralisant à d'autres services techniques que les services de transactions mais elle propose également une observation des variations du contexte d'exécution. Cette extension permet d'adresser l'adaptabilité des services techniques dans les environnements ubiquitaires. Ces environnements se caractérisent par une évolution dynamique de l'architecture du système d'informations et des capacités de traitements très hétérogènes. Ainsi, cette approche propose de compléter la description du service de transactions avec la prise en compte des informations liées à la description de la qualité de service. Ces informations sont utilisées par un service d'adaptation pour sélectionner le service technique le plus adapté aux besoins d'une application du point de vue de ses propriétés techniques et à l'adéquation de ses capacités de traitement.

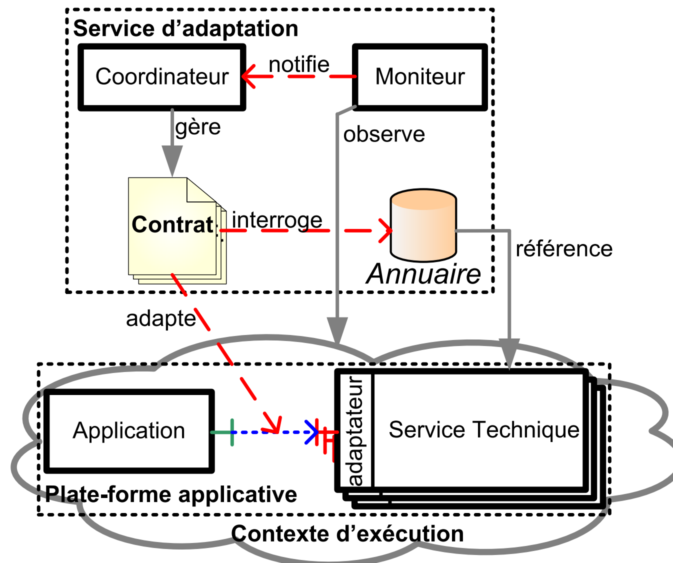


FIG. 2.18: Architecture du service d'adaptation.

Le cadre de conception défini dans cette approche repose également sur les principes de conception par composants et elle exploite les capacités du modèle de composants FRACTAL [BCL+06] (présenté dans la section 3.3.1 du chapitre suivant). Dans cette architecture présentée dans la figure 2.18, le composant Coordinateur constitue la clé de voûte du service d'adaptation car il maintient la cohérence entre les **Contrats** et le Moniteur du système. Ce médiateur supporte différents modes de propagation des informations selon que cette propagation est à l'initiative du contrat (mode *pull*) ou du moniteur (mode *push*). Le **Contrat** garantit quant à lui la cohérence entre les besoins d'une Application et le `Service Technique` qui répond au mieux à ces besoins. Le Moniteur se charge d'observer le *contexte d'exécution* de

l'application et utilise un mécanisme de filtrage des informations pour limiter les notifications du coordinateur aux événéments pertinents. Lorsque qu'une variation notable du système est détectée par le Moniteur, celui-ci notifie le `Coordinateur` des nouveaux paramètres du contexte d'exécution. Ces paramètres sont transmis au(x) **Contrat(s)** affecté(s) par la variation du contexte. Le **Contrat** interroge l'Annuaire pour obtenir la référence du service technique adapté au nouveau contexte d'exécution. Cet Annuaire intègre des fonctions de nommage et de courtage pour offrir un mécanisme de sélection des services techniques selon leurs propriétés techniques et leurs capacités de traitement. Si un nouveau Service Technique nécessite d'être associé à l'Application, alors le **Contrat** reconfigure l'assemblage de l'application avec le service technique. Pour ce faire, le service d'adaptation reconfigure le contrôleur de l'application en remplaçant l'instance de service technique existante par la nouvelle instance et en connectant cette dernière aux composants façade et *callback* du contrôleur.

Synthèse. Le service d'adaptation adresse l'**adaptabilité** des services techniques dans les applications à base de composants. La **granularité** d'adaptation supportée se situe au niveau du service technique. Pour ce faire, le service d'adaptation est **configurable** à l'aide d'un *modèle de description de la qualité de service* des services techniques disponibles. Ces descriptions alimentent un mécanisme de courtage adapté aux services techniques. Ce mécanisme de courtage est utilisé par un `Coordinateur` qui se charge de reconfigurer l'application lorsque le `Moniteur` détecte une variation notable du contexte d'exécution. L'**intégration** du service technique au niveau de l'application est alors supportée par le contrôleur et les composants façade et *callback*. La **modularité** du service d'adaptation est basée sur l'application de la *conception par composants*. Enfin, les **performances** et le coût d'utilisation du service d'adaptation ne sont pas abordés dans les travaux de [Hér05], cependant le coût de la reconfiguration du contrôleur de l'application s'avère manifestement pénalisant pour les performances de l'application. De plus, l'intérêt de l'approche réside dans la capacité du service d'adaptation à choisir un service de transactions approprié parmi une collection de services de transactions disponibles. Cependant, le nombre de services de transactions compatibles et présentant des caractéristiques différentes demeure limité à l'heure actuelle.

2.3.6 *Arjuna Transaction System*

Présentation *Arjuna Transaction System* (ARJUNATS) est issu des travaux initiés par l'université de Newcastle sur l'étude de l'utilisation du paradigme de programmation par objets pour la construction d'intergiciels tolérants aux fautes [PSWL95, LS02, Lit05]. ARJUNATS fait partie de l'environnement de programmation *Arjuna* [SDP91] dans lequel les actions atomiques imbriquées (*atomic nested actions*) constituent le modèle de calcul élémentaire pour contrôler les opérations de manipulation des objets persistants du système. À partir de cette solution généraliste, les implantations *OTSArjuna* et *JTSArjuna* ont été réalisées pour répondre respectivement aux spécifications transactionnelles *Object Transaction Service* [OMG03] et *Java Transaction Service* [Che99]. Plus récemment, la solution *Arjuna Transaction System* a également été déclinée pour réaliser les spécifications *Web Services Atomic Transaction* [Arj03b, CCF+05a] et *Business Activity* [CCF+05c]. La solution ARJUNATS a fait preuve de robustesse et de fiabilité face à l'évolution du domaine transactionnel et fait désormais partie intégrante du serveur d'application J2EE JBOSS [FR03].

L'architecture de ARJUNATS est organisée en plusieurs modules afin de faciliter la gestion de ses fonctionnalités. Cette modularité repose sur la définition d'interfaces de programmation indépendantes des différentes réalisations que peut fournir ARJUNATS. Basiquement, les modules disponibles dans ARJUNATS sont les modules *Atomic Action*, *RPC*, *Object Store* et *Naming and Binding*. Cependant, ARJUNATS se veut extensible et supporte la définition de nouveaux modules et l'extension des notions de base définies.

ARJUNATS repose principalement sur les notions de State Manager, Atomic Action, Lock Manager et Abstract Record présentées dans la figure 2.19 et représentant les abstractions suffisantes

des systèmes transactionnels existants. Ceci signifie que ces notions peuvent être dérivées pour supporter les standards transactionnels OTS, JTS et WS-AT sans remettre en cause l'architecture interne de ARJUNATS. Le gestionnaire d'état (State Manager) gère l'état persistant des objets du système en fournissant des opérations génériques pour (dés)activer les objets (`activate()` et `deactivate()`) et sauvegarder/restaurer leur état en mémoire persistante (`save_state()` et `restore_state()`). L'action atomique (Atomic Action) étend le gestionnaire d'état et fournit les opérations de démarcation transactionnelle (`begin()`, `commit()` et `abort()`) utilisée par l'application pour délimiter des sections critiques correspondant à des accès aux objets persistants du système. Le gestionnaire de verrous (Lock Manager) contrôle les accès concurrents aux objets persistants du système en associant des verrous typés (lecture, écriture, etc.) avec une action atomique en cours d'exécution. La sémantique des verrous (Lock) n'est pas imposée par le gestionnaire de verrous, ce qui permet à ARJUNATS de réaliser différentes politiques de verrouillage des données. L'enregistrement abstrait (Abstract Record) étend également le gestionnaire d'états et correspond à la représentation des objets du système manipulés par l'action atomique. Cette représentation dialogue avec l'action atomique lors de l'exécution du protocole de validation en deux phases via les opérations (`topLevelPrepare()`,`nestedPrepare()`,`phase2Commit()` et `phase2Abort()`).

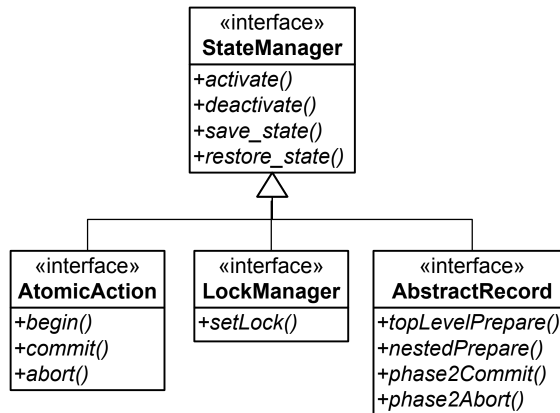


FIG. 2.19: Abstraction fonctionnelle définies par ARJUNATS.

Synthèse. ARJUNATS est un canevas extensible pour la construction de services de transactions fiables. La **modularité** du système repose sur l'identification de quatre modules *Atomic Action*, *RPC*, *Object Store* et *Naming and Binding* permettant d'isoler les fonctionnalités du système. Ainsi, ARJUNATS applique les principes de la *programmation par objets* pour offrir une implantation modulaire des fonctionnalités requises par la majorité des services de transactions. ARJUNATS fournit une architecture minimaliste d'un service de transactions reposant sur quatre notions de base pertinentes : State Manager, Atomic Action, Lock Manager et Abstract Record. Par conséquent, la **granularité** de ARJUNATS dépend des objets réifiant les notions de base du système. ARJUNATS peut être **étendu** pour définir de nouvelles politiques de contrôle de concurrence ou de reprise sur fautes. Cependant, la configuration de ARJUNATS est limitée à un fichier de configuration détaillant les caractéristiques du service de transactions, c.-à-d. la nature des classes implantant les notions de base du système. L'**intégration** d'ARJUNATS a été validée par la réalisation de différentes spécifications transactionnelles (OTS, JTS, WS-AT). Enfin, ARJUNATS est l'un des services de transactions les plus fiables et **performants** sur le marché à l'heure actuelle.

2.3.7 Autres approches

Cette section présente succintement quelques travaux relatifs à nos préoccupations et qui ont abordé l'intégration des aspects transactionnels dans les intergiciels à base de composants.

Modèle de transactions *Open Nested Transactions*

Présentation. Les travaux présentés dans [Nem04, HNL04] abordent la définition et l'implantation d'un nouveau modèle de transactions étendu. Ce nouveau modèle de transactions, appelé *Open Nested Transactions* (ONT), se veut particulièrement adapté aux transactions utilisées par les applications de commerce électronique. Celui-ci combine un modèle de transactions emboîtées [GR93] et un mécanisme de compensation [GMS87] pour annuler les effets d'une transaction validée. Dans ce modèle, les effets des sous-transactions validées sont visibles entre les différentes branches de la transaction racine en relachant la propriété d'isolation. La validation repose sur la propriété d'atomicité sémantique permettant de défaire les effets d'une transaction préalablement validée en cas d'abandon de sa transaction mère.

Synthèse. L'implantation de ce modèle de transactions étendu est réalisée en appliquant les *principes de la programmation par composants* et utilise en particulier le modèle de composants Fractal [BCL+06]. Cette implantation réutilise le service de transactions JOTM [Mes03] sous la forme d'un composant monolithique et définit de nouveaux composants dédiés à l'implantation du modèle ONT. Le service de transactions résultant est peu adaptable dans la mesure où son objectif est de fournir uniquement une implantation du modèle ONT et n'adresse ni son intégration dans les applications ni la prise en compte de l'adaptation de l'intergiciel.

Politiques de démarcation BOURGOGNETS

Présentation. Les travaux réalisés autour du canevas intergiciel BOURGOGNETS adressent l'intégration des modèles de transactions étendus dans les plates-formes à base de composants [PP01, Pro01, Pro02]. Pour ce faire, BOURGOGNETS étend les politiques de démarcation communément définies dans les plates-formes à composants. Ces politiques de démarcation permettent de contrôler automatiquement l'exécution et la validation des transactions au niveau des méthodes des composants hébergés par la plate-forme.

Plutôt que d'utiliser le descripteur de déploiement de la plate-forme applicative, BOURGOGNETS propose également de spécifier le comportement de démarcation transactionnelle directement au niveau de la définition des méthodes des interfaces fournies par le composant. Cette définition repose sur la spécification du comportement transactionnel à appliquer au regard de l'exécution en cours ou non d'une transaction lors de l'invocation d'une méthode du composant.

Synthèse. Le support des modèles de transactions étendus requiert également l'extension du service de transactions associé à la plate-forme applicative afin de prendre en charge la gestion des modèles de transactions étendus. Pour ce faire, BOURGOGNETS propose un ensemble d'interfaces dédiées à l'implantation des modèles de transactions étendus supportés par le modèle générique BOURGOGNE. Ces interfaces viennent compléter les interfaces définies par les services de transactions traditionnels tels que JOTM [Mes03]. Cependant, cette extension repose fortement sur le service de transactions sous-jacent.

2.4 Synthèse

Les travaux évoqués dans ce chapitre présentent les différents aspects adressés par le domaine transactionnel. Nous avons vu que ce domaine est sujet à une grande disparité des modèles de transactions, des protocoles de validation et des standards transactionnels. Pour faire face à cette disparité, de nombreux travaux ont tenté de proposer une solution originale pour fournir un support d'exécution adaptable aux besoins des applications. Le tableau 2.1 présente une synthèse de l'évaluation des critères que nous avons jugé pertinent pour la construction de services de transactions adaptables. Pour chacun des travaux que nous avons abordé, nous y résumons les éléments de solution proposés par ces approches face aux critères que nous avons définis. Nous

utilisons le caractère **+** pour identifier une propriété intéressante et le caractère **–** pour décrire une faiblesse de l'approche observée.

Reflective Transaction Framework propose une granularité d'adaptation très fine en définissant un protocole à méta-objets supportant l'adaptation comportementale des différentes méthodes disponibles sur un service de transactions existant. Cette approche à granularité très fine permet de modifier le comportement d'un service de transaction existant de manière transparente, mais il introduit également un surcoût à l'exécution non négligeable lié à la manipulation du méta-protocole transactionnel.

Kernel Aspect Language for ATMS (KALA) propose non seulement une approche déclarative pour simplifier le support des modèles de transactions étendus au niveau applicatif mais il définit également une abstraction fonctionnelle du service de transactions supportant ces modèles de transactions étendus en s'inspirant du canevas ACTA. Cette abstraction fonctionnelle peut être assimilée à un langage dédié à la configuration d'un modèle de transactions étendu. Cependant, KALA repose sur l'utilisation d'un service de transactions spécifique dont les interfaces sont très liées au langage de configuration du modèle de transactions étendu.

REFLECTS propose de traiter l'adaptation des modèles de transactions à l'exécution en définissant un environnement de déploiement et de courtage sémantique basé sur une architecture à composants. Cette approche introduit un mécanisme d'adaptation dynamique du service de transactions en fonction des besoins applicatifs. Cependant, la granularité de cette approche n'autorise que l'adaptation des services de transactions et ne permet pas de reconfigurer des caractéristiques plus fines (par exemple, le protocole de validation).

Le *service d'adaptation* propose également de traiter l'adaptation des modèles de transactions à l'exécution et de façon transparente pour l'application en définissant un modèle de qualité de service et en utilisant des sondes pour détecter des variations dans le contexte d'exécution et adapter le service de transactions utilisé par l'application en conséquence. Dans cette approche comparable à REFLECTS, la sensibilité au contexte est prise en compte par une notion de qualité de service associée au service de transactions. Cependant, le coût de l'observation du contexte vient s'ajouter au coût du mécanisme de courtage nécessaire à la sélection du service de transactions le plus approprié au contexte d'exécution courant. De plus, l'intérêt de cette approche repose, comme REFLECTS, sur la disponibilité d'un nombre suffisant de services de transactions aux caractéristiques différentes.

Arjuna Transaction System (ARJUNATS) propose de réaliser un service de transactions non seulement performant et fiable, mais qui peut également implanter différents standards transactionnels. Pour ce faire, ce service de transactions définit des abstractions fonctionnelles efficaces qui lui sont propres et qui permettent de modulariser sa structure. Cependant, la configuration d'ARJUNATS est limitée par l'utilisation d'une approche orientée objet et d'un mécanisme de descripteur de configuration.

Open Nested Transactions propose non seulement un nouveau modèle de transactions mais il structure également son implantation sous la forme d'un ensemble de composants patrimoniaux ou dédiés. Cette structuration sous forme de composants permet de définir différentes configurations de services de transactions afin qu'ils supportent des modèles de transactions différents (ici le modèle de transactions plates ou emboîtées ouvertes). Cette approche représente l'implantation d'un modèle de transactions étendu particulier en utilisant un approche de conception par composants à gros grain ne supportant que l'adaptation du service de transaction.

Bourgogne Transaction Service propose de compléter et de simplifier la démarcation des transactions dans les plates-formes applicatives à base de composants en remplaçant la spécification des politiques de démarcation dans les descripteurs de déploiement par une spécification au niveau des interfaces métiers des composants applicatifs. Ce canevas offre la possibilité de supporter de nouvelles politiques de démarcation dans les plates-formes

Critères	OUVERTURE		GRANULARITÉ		MODULARITÉ	
REFLECTIVE TF	modèles de transactions étendus	+	méthode	++	méta-objets	−
KALA	modèles de transactions étendus	+	primitive KALA	+	aspects	+
REFLECTS	services de transactions	−	service	−	composants	++
Service d'adaptation	services techniques	−	service	−	composants	++
ARJUNATS	standards transactionnels	+	objet	+	interfaces	+
ONT	N/A		service	−	composants	++
BOURGOGNETS	politiques de démarcation	+	objet	+	N/A	
GOTM	transactions, services, standards et démarcation	++	interface de fine granularité	++	composants	++

Critères	PERFORMANCE		CONFIGURATION		INTÉGRATION	
REFLECTIVE TF	surcoût lié à la réflexion	−	méta-descripteur	+	interface dédiée	−
KALA	aucun surcoût	++	langage dédié	+	interfaces dédiées	−
REFLECTS	surcoût lié à la sélection	-	descripteur de caractéristiques	+	interfaces dédiées	−
Service d'adaptation	surcoût lié à la sélection et à l'observation	−	descripteur de qualité de service	+	composant façade	+
ARJUNATS	aucun surcoût	++	descripteur de propriétés	-	interfaces standardisées	++
ONT	N/A		N/A		interfaces dédiées	−
BOURGOGNETS	N/A		N/A		interfaces dédiées	−
GOTM	aucun surcoût	++	modèles de haut niveau	++	interfaces standardisées	++

TAB. 2.1: Synthèse de l'adaptabilité dans les intergiciels transactionnels.

à base de composants EJB. Cependant, la solution proposée par cette approche est très intrusive au niveau du développement de l'application de par la définition d'un jeu d'interfaces de démarcation transactionnelle spécifiques et l'utilisation d'un formalisme dédié à la démarcation au niveau des interfaces applicatives.

Tous ces travaux abordent différents aspects d'adaptation des services de transactions (modèles de transactions, standards, etc.) que ce soit lors de la conception du service ou lors de son exécution. Cependant, peu de ces travaux (à l'exception d'ARJUNATS) n'adressent les aspects de performance et de support des standards qui sont pourtant critiques dans la construction des intergiciels modernes. En outre, aucun de ces travaux ne propose un cadre uniforme pour supporter plusieurs aspects d'adaptation des services de transactions, et la diversité des approches actuelles ne permet aucune combinaison des propositions pour aboutir à une solution unique.

2.5 Conclusion

Dans ce chapitre, nous avons présenté les différentes notions généralement associées au domaine du transactionnel. Depuis leur introduction dans les bases de données, les notions liées aux transactions n'ont jamais cessé d'évoluer pour s'adapter aux besoins des applications. Dans le domaine des intergiciels, différents travaux ont adressé la conception de services de transactions adaptables. Nous avons présenté puis évalué six intergiciels transactionnels supportant différents types d'adaptations. La synthèse établie à l'issue de ces évaluations souligne le fait que les approches actuelles se limitent à un seul aspect d'adaptation des services de transactions sans fournir de solution homogène pour la construction de services de transactions hautement adaptables.

Au regard des contributions et des limites des travaux existants, nous décidons de baser notre approche sur la construction d'un canevas logiciel extensible pour implanter différents services de transactions (comme dans ARJUNATS). Ce canevas repose sur une approche de conception par composants afin de modulariser la structure des services de transactions en différents composants (comme dans REFLECTTS, *service d'adaptation* ou ONT). Les interactions entre les composants sont identifiées par des interfaces qui correspondent aux abstractions fonctionnelles de notre canevas (comme dans ARJUNATS). Afin de maximiser les possibilités d'ouverture des services de transactions construits avec ce canevas, nous proposons de concevoir des composants de très fine granularité (comme dans *Reflective Transaction Framework*). Nous souhaitons proposer un formalisme de configuration compréhensible par l'utilisateur (comme dans KERNEL ASPECT LANGUAGE FOR ATMS). Enfin, nous attachons un intérêt particulier aux performances des services de transactions résultants afin de démontrer qu'une approche favorisant l'adaptation aux besoins de l'application n'introduit pas nécessairement de surcoût à l'exécution (comme dans ARJUNATS).

The maintenance engineer will never have seen a model quite like yours before.

Murphy's Law

Chapitre 3

Modèles de composants pour la construction de canevas intergiciels

Sommaire

3.1 Introduction 37
3.2 Prérequis 38
3.2.1 Concepts fondamentaux 38
3.2.2 Conception à base de composants 39
3.2.3 Architectures logicielles 40
3.2.4 Canevas intergiciels 40
3.3 Étude de l'existant 41
3.3.1 Modèle de composants FRACTAL 41
3.3.2 Modèle de composants OPENCOM 46
3.3.3 Modèle de composants ABC 49
3.3.4 Autres approches 53
3.4 Conclusion 56

CE CHAPITRE PRÉSENTE UNE ÉTUDE DES MODÈLES DE COMPOSANTS *utilisés dans le développement de canevas intergiciels. Pour ce faire, nous procédons dans un premier temps à une présentation des concepts fondamentaux du génie logiciel qui ont motivé la définition des modèles de composants (cf. section 3.2). Nous introduisons également les notions d'architecture logicielle et de canevas intergiciel comme des concepts importants pour notre proposition.*
Ce chapitre se poursuit avec l'étude de différents modèles de composants légers et de leur langage de description d'architectures dont l'utilisation a pu être validée sur le développement d'intergiciels adaptables (cf. section 3.3). Enfin, après avoir synthétisé les avantages et inconvénients de ces différentes approches, nous motivons notre choix pour le modèle de composants FRACTAL *tout en présentant les différentes extensions que nous souhaitons lui apporter (cf. section 3.4).*

3.1 Introduction

Les composants sont désormais reconnus comme une approche fiable et efficace pour le développement des couches applicatives et intergicielles. Les propriétés de modularité et d'abstraction qu'ils offrent permettent notamment de maîtriser le développement d'intergiciels complexes. Cette maîtrise est renforcée par l'utilisation des langages de description d'architecture

qui fournissent un mécanisme simple de configuration des personnalités des canevas intergiciels. Ces personnalités correspondent à des configurations déterminées des canevas intergiciels intégrables par une application afin que cette dernière puisse interagir avec le système d'exploitation via l'intergiciel.

La suite de ce chapitre est organisée de la manière suivante. La section 3.2 introduit non seulement les concepts fondamentaux du génie logiciel et les notions de modèles de composants, d'architecture logicielle et de canevas intergiciel. La section 3.3 présente une étude des modèles de composants FRACTAL, OPENCOM et ABC dédiés au développement des intergiciels. Enfin, la section 3.4 conclut cet état de l'art en motivant notre choix de modèle de composants et les différentes extensions qu'il est nécessaire de définir dans le cadre de notre approche.

3.2 Prérequis

Cette section procède à l'introduction des concepts liés à la conception orientée composants et aux architectures logicielles dans le contexte du développement des intergiciels.

3.2.1 Concepts fondamentaux

Cette section présente quelques concepts fondamentaux usuellement évoqués dans le domaine du génie logiciel et de la construction des intergiciels. Ces notions correspondent à des principes de conception reconnus et permettant de maîtriser la complexité d'un logiciel.

Séparation des préoccupations

La séparation des préoccupations (en anglais, *Separation of Concerns*) est un concept fort du génie logiciel. Ce principe, dont nous devons la première citation à Descartes, vise à « *Décomposer les difficultés en autant de parcelles qu'il se peut pour mieux les résoudre* ». Dans le domaine du génie logiciel, cette séparation des préoccupations peut prendre différentes formes quelles soient organisationnelles (par exemple, découpage des tâches de conception dans le temps ou selon les acteurs) ou structurelles (par exemple, découpage du code applicatif). Néanmoins, ce concept vise à maîtriser la complexité d'un système en le divisant en un certain nombre de préoccupations. Les préoccupations correspondent alors à des concepts, des vues, ou des sujets communs apparaissant de façon récurrente dans une application. Par conséquent, la complexité dépend en partie de l'organisation des interactions au sein ou entre les préoccupations d'un système.

La séparation des préoccupations est également à l'origine du principe d'orthogonalité. Ainsi, une préoccupation est dite orthogonale si elle minimise son couplage avec les autres préoccupations impliquées dans la conception de l'application. Pour éviter que les préoccupations ne nuisent à la pérennité d'une application, le concept de *module* a été introduit pour traiter de la séparation des préoccupations structurelles.

Modularité

La notion de modularité est issue des travaux initiés autour de la programmation modulaire [Par72]. La modularité vise à fournir un découpage structurel d'une application en différents modules. Les modules se caractérisent par une forte cohésion interne et un faible couplage entre les modules. Cependant, le principe de modularité ne définit pas précisément la nature du contenu des modules (code, données, modèle, etc.) mais il préconise que les modules doivent être des élements de structure simples. Grâce à l'application de ce principe, il est possible d'accroître la productivité en utilisant les modules comme des éléments de structure réutilisables.

Le premier modèle de programmation modulaire [Par72] a notamment permis de regrouper un ensemble de fonctions adressant une même préoccupation. Puis, le modèle de programmation par objets [DEMN98] a pris en compte non seulement les fonctions mais également les

données mises en jeu par une préoccupation. Ensuite, le modèle de programmation par composants [SGM02] a permis d'identifier les interactions entre les modules d'une application (appelés composants) en établissant la notion de contrat entre deux composants ayant une relation de dépendance. Finalement, le modèle de programmation par aspects [KLM+97] a permis d'isoler différentes préoccupations dites orthogonales. L'intégration des aspects dans une application est réalisée *a posteriori* par l'utilisation d'un tisseur et d'un langage de coupe décrivant les points d'interaction entre l'application et les aspects à intégrer. Ces différentes évolutions de la notion de modularité ont permis de mieux identifier et isoler les préoccupations qui peuvent être impliquées dans le développement d'une application.

Abstraction

Le principe d'abstraction adresse également la complexité d'une application [GJM02]. Ce principe propose de masquer incrémentalement les détails d'une préoccupation donnée en fournissant une vue plus simplifiée du système. La maîtrise de ce concept permet de représenter une même préoccupation avec différents degrés de détails tout en adaptant cette vue aux besoins du concepteur. Une bonne abstraction du système doit être autant que possible orthogonale aux autres abstractions afin de pouvoir être facilement isolée et composée.

3.2.2 Conception à base de composants

Ces dernières années, les approches à base de composants se sont imposées comme un paradigme de conception puissant pour le développement des applications. L'émergence de ce paradigme a notamment permis d'améliorer le passage à l'échelle, l'administration et la pérennité des systèmes répartis en s'appuyant sur les principes de séparation des préoccupations, de modularité et d'abstraction des couches intergicielles. Ainsi, les modèles de composant s industriels tels que CCM (*Corba Component Model*) [OMG04] , EJB (*Enterprise Java Beans*) [DK05] ou SCA (*Service Component Architecture*) [IBM05] ont pu être validés industriellement sur le développement de larges applications réparties.

Les concepts fondamentaux liés aux composants ont peu évolué depuis qu'il ont été introduits par McIlroy [McI68]. Néanmoins, ils ont été récemment formalisés dans la définition suivante :

> *A component is a unit of composition with* contractually specified interfaces *and* context dependencies *only. A software component can be* deployed *independently and is subject to* composition *by third parties.*
>
> *Component Software : Beyond Object-Oriented Programming* [SGM02]

Le succès de ce paradigme a suscité un intérêt croissant de la communauté de développement des intergiciels afin de trouver une solution aux problèmes d'évolution des plates-formes intergicielles modernes. Cependant, les modèles de composant s définis pour les applications ne sont pas nécessairement adaptés au développement des intergiciels. En particulier, les intergiciels recherchent des modèles de composants suffisamment légers et performants pour ne pas pénaliser les couches logicielles construites au dessus. Dès lors, de nombreuses propositions de modèles de composants légers ont émergé pour répondre aux besoins des intergiciels. Ces modèles de composants définissent les concepts fondamentaux pour bénéficier des avantages de la programmation par composants tout en limitant ses inconvénients. En particulier, ces modèles de composants légers n'imposent pas un support figé des propriétés techniques des composants (par exemple, transactions et sécurité) mais proposent des solutions beaucoup plus flexibles pour adapter le conteneur aux besoins du composant. Le conteneur représente le support d'exécution d'un composant logiciel et lui fournit les différents services implantés par la plate-forme logicielle. Le support des composants est complété afin de pouvoir introspecter et reconfigurer les applications dynamiquement. De plus, les modèles de composants légers utilisent de plus en plus les notions de hiérarchie et de partage afin d'accroître la modularité des systèmes. Enfin, pour respecter le principe de séparation des préoccupations et afin de maîtriser la complexité

grandissante des intergiciels, les modèles de composants s'appuient également sur un langage de description d'architectures afin de construire les intergiciels par composition des briques élémentaires que représentent les composants.

3.2.3 Architectures logicielles

L'architecture logicielle a joué un rôle primordial dans la réalisation d'applications complexes. Cette réflexion sur l'architecture d'une application permet notamment d'anticiper les problèmes d'évolution et d'assurer la pérennité du système. Les architectures logicielles ont suivi une ascension similaire aux approches par composants, et ce grâce à leur complémentarité. Dans ce domaine, la notion de *langage de description d'architectures* (ADL) (en anglais, *Architecture Description Language*) est prépondérante [MT00]. Le langage de description d'architectures correspond à un langage dédié à la description structurelle des modules impliqués dans la réalisation d'une application.

Il existe de nombreux langages de description d'architecture tels que DARWIN, ACME, WRIGHT, OLAN, AADL, RAPIDE, UNICON, XADL ou C2 [MT00, Bar05]. Néanmoins, une définition reconnue de la notion d'architecture logicielle est la suivante :

> *A software architecture of a program or computing system is the structure or* structures *of the system, which comprise* software components*, the externally* visible properties *of those components, and the* relationships *among them.*
>
> ---
>
> *Software Architecture in Practice* [BCK03]

Les concepts communément associés au domaine des architectures logicielles sont [MT00] :
- les *composants* qui représentent les entités élémentaires d'une application,
- les *connecteurs* qui identifient les types d'interactions entre les composants de l'architecture,
- les *configurations* qui décrivent une architecture en terme de composants et de connecteurs,
- les *composites* qui réifient une configuration sous la forme d'un composant.

Bien que la notion d'architecture logicielle demeure indépendante des approches de conception orientée composants, leur association permet de multiples applications. Celles-ci adressent non seulement la vérification des configurations décrites, mais également la génération d'un exécutable et son déploiement voire sa reconfiguration.

3.2.4 Canevas intergiciels

Les canevas intergiciels (en anglais, *middleware framework*) représentent une nouvelle forme de bibliothèque logicielle facilitant la construction des intergiciels adaptables comme illustré dans la figure 3.1. Un canevas intergiciel capitalise les fonctionnalités communes au domaine qu'il considère dans un ensemble de composants patrimoniaux. Ces composants peuvent être facilement réutilisés pour construire de nouvelles formes d'intergiciels communément appelées **personnalités d'intergiciel**. Pour ce faire, le canevas intergiciel s'appuie sur un langage de description d'architectures qu'il utilise en complément du modèle de composants pour décrire la configuration d'une personnalité. Une personnalité fournit une abstraction du **système d'exploitation** adaptée aux besoins d'une **application**. Ces canevas permettent de développer rapidement des couches logicielles fiables utilisables par une application requiérant un support de la persistance des données, de la tolérance aux fautes ou des communications distantes.

La figure 3.2 présente deux exemples de canevas intergiciels existants dédiés à la communication et à la persistance. Le canevas JONATHAN, présenté dans la figure 3.2a, est un canevas générique de communication servant à construire différentes personnalités de bus de communications tels que des bus CORBA, JavaRMI ou FractalRMI [DHTS99]. Les personnalités construites au dessus de JONATHAN peuvent être utilisées dans des projets tels que JOTM [Mes03], JONAS [Exe04] ou FRACTAL [BCL+06]. Les canevas JORM, MEDOR et PERSEUS, présenté dans la figure 3.2b, sont des canevas élémentaires qui peuvent être combinés pour réaliser une personnalité de support de la persistance telle que *Java Data Objects* (JDO) ou la persistance

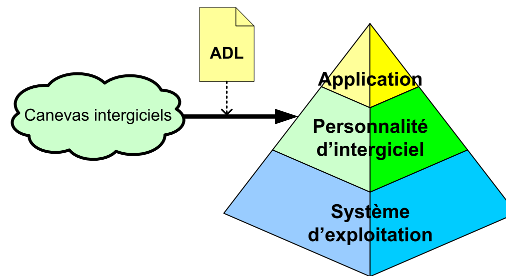
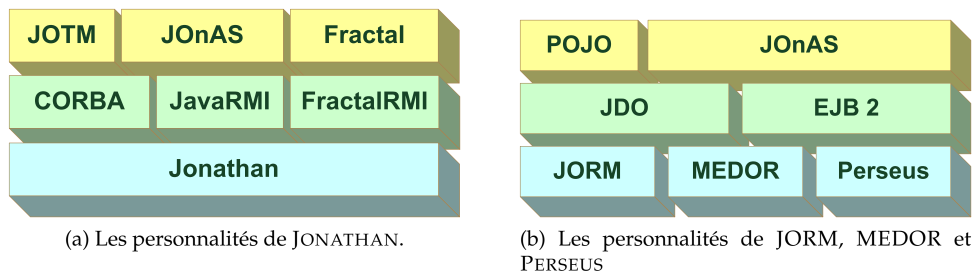


FIG. 3.1: Positionnement du canevas intergiciel dans les applications modernes.

EJB 2 [ACBD+04]. Ces personnalités peuvent être utilisées par des serveurs d'applications tels que JOnAS ou par de simples applications Java (POJO) dans le cas de la personnalité JDO.



(a) Les personnalités de JONATHAN.

(b) Les personnalités de JORM, MEDOR et PERSEUS

FIG. 3.2: Exemples de personnalités de canevas intergiciels existants.

3.3 Étude de l'existant

Cette section étudie trois modèles de composants légers (FRACTAL, OPENCOM et ABC) qui ont déjà été utilisés dans le cadre de développements d'intergiciels adaptables. Notre étude observe non seulement les caractéristiques de ces modèles de composants mais aussi les langages de description d'architecture qui leurs sont associés, et des exemples d'utilisation de ces modèles sur des applications de taille réelle. Cette étude est ensuite complétée par une étude du langage de programmation de composants ARCHJAVA et du langage de description d'architectures DARWIN.

3.3.1 Modèle de composants FRACTAL

Le modèle de composants FRACTAL a été défini par France Telecom R&D et l'INRIA et se présente sous la forme d'une spécification [BCS04] et d'implantations dans différents langages (JULIA et AOKELL en Java, Think en C, Plasma en C++, Fractalk en Smalltalk).

Le modèle de composants Fractal est déjà appliqué à tous les niveaux du logiciel : des systèmes d'exploitation (par exemple, THINK [FSLM02]), aux intergiciels (par exemple, DREAM [Qué05]) et aux serveurs d'application (par exemple, JONAS à la carte [Abd06]).

Le modèle de composants hiérarchique FRACTAL est basé sur les notions de *composant*, *interface* et *liaison* [BCL+04, BCL+06]. Ce modèle de composants dispose d'un modèle mathématique offrant une base formelle solide pour la construction d'intergiciels fiables et performants [HHP+05]. Un composant est une entité exécutable conforme au modèle FRACTAL. Un composant primitif encapsule une unité de calcul décrite dans un langage de programmation déterminé. Une interface est un point d'interaction exprimant les méthodes requises ou fournies par un composant. Une liaison est un canal de communication établi entre des interfaces.

De plus, Fractal supporte le *partage récursif* et le *contrôle réflexif* [BCS02]. Le partage récursif implique qu'un composant composite peut être composé de plusieurs sous-composants à n'importe quel niveau et qu'un composant peut être contenu par plusieurs composants. Le contrôle réflexif implique qu'une architecture construite avec Fractal est réifiée à l'exécution, et peut être introspectée et manipulée dynamiquement. Ce type de manipulation est supporté par un ensemble de contrôleur et d'intercepteurs qui forment la membrane du composant. Le nombre et la nature de ces interfaces ne sont pas figés mais peuvent être étendus par opposition aux modèles de composants industriels tels que EJB ou CCM. Cette membrane peut être étendue via le modèle de programmation des contrôleurs basé sur l'usage des *mixins* [BCL+06] ou via des aspects [SPDC06].

La figure 3.3 décrit les différentes entités mises en jeu dans une architecture de type FRACTAL. Le rectangle noir représente le contrôle du composant alors que l'intérieur du rectangle représente le contenu du composant. Les flèches correspondent aux liaisons tandis que les formes en T attachées aux rectangles correspondent aux interfaces internes ou externes du composant. Les interfaces internes ne sont accessibles que depuis l'intérieur du composant. Les interfaces externes apparaissant au dessus du composant sont les interfaces de contrôle telles que le contrôleur de composant (c), le contrôleur de cycle de vie (lc), le contrôleur de liaisons (bc), le contrôleur de contenu (cc) ou le contrôleur d'attributs (ac).

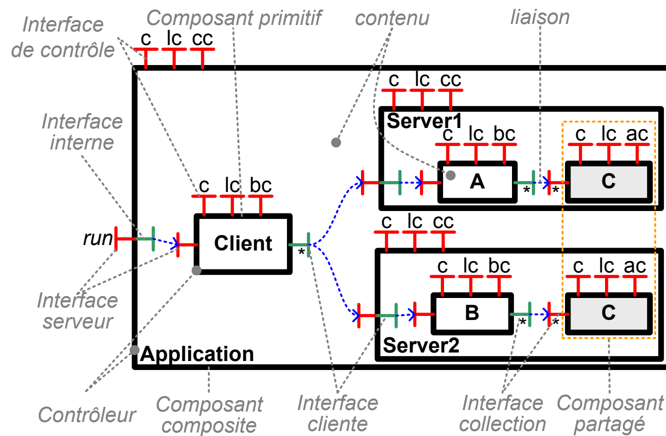


FIG. 3.3: Architecture à base de composants FRACTAL.

Le langage de description d'architectures FRACTAL ADL et son usine extensible

FRACTAL fournit également un langage de description d'architectures, appelé FRACTAL ADL, afin de permettre la description et le déploiement automatique de configurations à base de composants. Ce langage de description d'architectures comme l'usine de déploiement associée sont également suffisamment extensibles pour permettre la prise en compte de nouvelles préoccupations (par exemple, trace [Qué05], aspects [PSCD06]). Cette extensibilité permet également de prendre en charge des aspects additionnels tels que le déploiement réparti, la vérification ou l'analyse d'architectures construites à base de composants. Le langage FRACTAL ADL est basé sur la notation XML et définit un ensemble de modules isolant différents aspects de la description d'architectures (implantation, liaisons, attributs, localisation, etc.).

Le listing 3.1 présente un exemple d'utilisation du langage FRACTAL ADL pour décrire l'architecture de l'application présentée dans la figure 3.3. La balise `<definition>` (ligne 1) marque la définition de l'architecture du composant racine de l'architecture. La balise `<interface>` (ligne 2) déclare une interface sur le composant de type `server` (fournie) ou `client` (requise). La balise `<component>` (ligne 3) déclare une instance de composant imbriquée dans le composant racine. Enfin, la balise `<binding>` (ligne 8) permet d'établir une liaison entre une interface

```xml
1 <definition name="Application">
2   <interface name="run" role="server" signature="Runnable"/>
3   <component name="client" definition="Client"/>
4   <component name="server1">
5     <interface name="s" role="server" signature="Service"/>
6     <component name="a" definition="A"/>
7     <component name="c" definition="C"/>
8     <binding server="this.s" client="a.s"/>
9     <binding server="a.r"    client="c.r"/>
10  </component>
11  <component name="server2">
12    <interface name="s" role="server" signature="Service"/>
13    <component name="b" definition="B"/>
14    <component name="c" definition="serveur1/c"/>
15    <binding server="this.s" client="a.s"/>
16    <binding server="a.r"    client="c.r"/>
17  </component>
18  <binding client="this.run"               server="client.run"/>
19  <binding client="client.service-server1" server="server1.service"/>
20  <binding client="client.service-server2" server="server2.service"/>
21 </definition>
```

LST. 3.1: Description FRACTAL ADL du composant Application.

requise et une interface fournie de composant situé dans le même composant. Les différentes balises de la grammaire basic.dtd du langage FRACTAL ADL sont résumées dans le tableau 3.1.

Balise	Parent	Propriétés	Description	Contingence	Valeur initiale
definition	-	name extends arguments	nom du composant définition étendue liste des arguments du composant	obligatoire optionnel optionnel	- - -
component	definition ou component	name definition	nom du composant définition du composant	obligatoire optionnel	- -
interface	definition ou component	name signature role contingency cardinality	nom de l'interface signature Java de l'interface rôle de l'interface (server|client) contingence de l'interface (mandatory|optional) cardinalité de l'interface (singleton|collection)	obligatoire obligatoire optionnel optionnel optionnel	- - server mandatory singleton
binding	definition ou component	client server	nom de l'interface cliente nom de l'interface serveur	obligatoire obligatoire	- -
attributes	definition ou component	signature	signature de l'interface Java de contrôle des attributs	obligatoire	-
attribute	attributes	name value	nom de l'attribut valeur de l'attribut	obligatoire obligatoire	- -
controller	definition ou component	desc	descripteur de la membrane du composant	obligatoire	-
template-controller	definition ou component	desc	descripteur de la membrane du composant *template* associé au composant	obligatoire	-
content	definition ou component	class	classe Java d'implantation du composant	obligatoire	-

TAB. 3.1: Description du langage FRACTAL ADL.

De nouveaux modules peuvent étendre les modules disponibles du langage FRACTAL ADL pour intégrer des nouveaux mots-clés à la syntaxe de FRACTAL ADL. Le support de ces mots-clés est réalisé par une extension de l'architecture de l'usine de FRACTAL ADL. Cette architecture constituée de composants FRACTAL supporte trois niveaux d'intégration comme illustré dans la figure 3.4 :

Le composant loader est une chaîne de délégation qui analyse les définitions FRACTAL ADL et construit un arbre abstrait (AST) (en anglais, *Abstract Syntax Tree*) correspondant. Chaque mot-clé du langage FRACTAL ADL est ainsi transformé en un nœud de l'AST par le composant basic loader puis les autres composants en amont de la chaîne de la délégation peuvent effectuer des traitements de modification ou de vérification du contenu de l'AST.

Le composant compiler utilise l'AST généré par le composant loader pour définir un ensemble de tâches à exécuter (par exemple, création d'un composant, définition de la valeur d'un

attribut, établissement d'une liaison). Chaque nœud de l'AST est interprété en un ensemble de tâches à réaliser au niveau intergiciel selon sa sémantique.

Le composant builder définit un comportement concret pour les tâches créées par le composant compiler. Cette isolation du comportement permet de fournir différentes implantations des tâches afin de fournir par exemple un mode de déploiement statique (génération du code Java de déploiement de l'application) en plus du mode de déploiement dynamique.

Le composant scheduler exécute les tâches générées par le composant compiler en fonction de leurs dépendances.

Le composant factory correspond à une façade dont le rôle est de coordonner les invocations réalisées sur les autres composants.

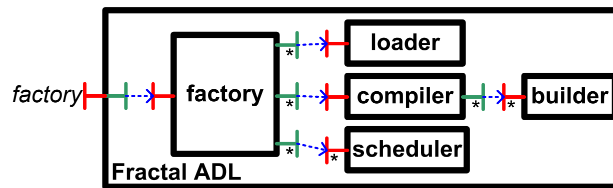


FIG. 3.4: Architecture de l'usine FRACTAL ADL.

L'architecture de l'usine FRACTAL ADL est par conséquent elle-même décrite en utilisant le langage FRACTAL ADL et son chargement est réalisé par une usine minimaliste préconfigurée. Le développement d'une extension de FRACTAL ADL et de son usine consiste donc à étendre la grammaire XML du langage et à ajouter de nouveaux composants dans l'architecture de l'usine.

Bibliothèque de composants DREAM

DREAM [LQS05, Qué05] est une bibliothèque de composants dédiée à la construction des intergiciels orientés messages (MOM) (en anglais, *Message-Oriented Middleware*). Les MOMs sont reconnus comme un moyen efficace pour construire des applications distribuées constituées d'entités faiblement couplées [BCSS99]. Ce canevas utilise le modèle de composants FRACTAL pour réifier les principaux concepts mis en jeu dans la définition d'un tel intergiciel — c.-à-d. les interfaces d'entrée et de sortie des messages, les gestionnaires de messages et les gestionnaires d'activités. Ces concepts sont ensuite exploités par la bibliothèque DREAM afin de fournir une implantation des différentes fonctions communément présentes dans les intergiciels orientés messages.

Les files de messages servent à stocker les messages. Celles-ci disposent d'une entrée (utilisée pour stocker les messages) et d'une sortie (utilisée pour délivrer les messages). Les stratégies associées à ces files diffèrent par la manière dont les messages sont triés (FIFO, LIFO, ordre causal, etc.), et leur comportement dans les différents états (file pleine, file vide, etc.).

Les transformateurs sont des composants disposant d'une entrée et d'une sortie. Chaque message reçu sur l'entrée est transformé puis délivré sur la sortie. Par exemple, un transformateur peut ajouter une information relative à une adresse IP dans le message.

Les pompes disposent d'une interface *pull* en entrée et d'une interface *push* en sortie. L'activité des pompes consiste à récupérer un message en entrée et à le délivrer sur la sortie.

Les routeurs ont une entrée et plusieurs sorties. Le rôle d'un routeur est d'orienter les messages reçus sur l'entrée vers une ou plusieurs sorties.

Les (dés)agrégateurs disposent de plusieurs entrées et d'une sortie (respectivement une entrée et plusieurs sorties). Leur rôle est d'agréger (et inversement) les messages.

Les canaux supportent les échanges de messages entre différents espaces d'adressage. Ceux-ci correspondent à des composants composites comportant au minimum deux composants : un canal sortant (en anglais, *channel out*) et un canal entrant (en anglais, *channel in*).

La figure 3.5 présente la ré-ingénierie de l'intergiciel orienté message JORAM [QBFL04] en utilisant la bibliothèque de composants DREAM. Dans cette architecture, le Moteur est composé d'une File de messages, d'un Protocole atomique pour l'exécution des agents et d'un Dépôt en charge de la création et de l'exécution des agents. Les composants Réseau 1 et Réseau 2 sont composés de Canaux entrant, et sortant et d'un transformateur pour résoudre l'adresse et le port de destination (Resolution destination). Le composant Réseau 1 comporte en plus un composant ordre causal pour garantir l'ordonnancement des messages et une File de messages pour découpler le flôt d'exécution du Moteur et du Réseau 1.

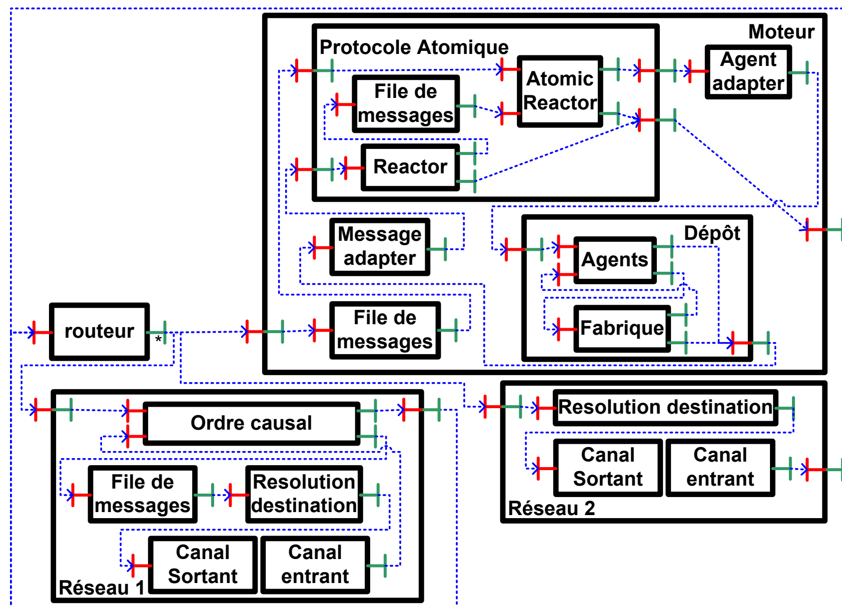


FIG. 3.5: Architecture à base de composants DREAM.

Synthèse

Le modèle de composants FRACTAL est un modèle de composants flexible dédié à la construction des intergiciels adaptables. Ce modèle de composants supporte la *définition d'architectures hiérarchiques avec partage*. Cette caractéristique facilite non seulement la séparation des préoccupations lors de la description des architectures mais il permet également de réduire l'empreinte mémoire de l'application en partageant certaines instances de composants. De plus, le modèle de composants FRACTAL fournit un *contrôle réflexif* modulaire et extensible permettant de choisir lors du déploiement de l'application entre une configuration statique à faible empreinte mémoire ou une architecture introspectable et fortement reconfigurable. Les performances des applications construites avec le modèle de composants FRACTAL et utilisant l'implantation JULIA ou AOKELL sont comparables aux applications équivalentes écrites directement en langage Java. En effet, le surcoût constaté à l'exécution dépend des intercepteurs de contrôle placés sur les composants (par exemple, l'intercepteur de contrôle du cycle de vie). Cependant, le modèle de programmation de FRACTAL est basé sur la réalisation de différentes interfaces de contrôle requises par le modèle FRACTAL (par exemple, `LifeCycleController`, `BindingController`) et le code associé à ces interfaces s'avère verbeux et n'est pas forcément intuitif à écrire.

Notre intérêt pour le langage de description d'architectures FRACTAL ADL porte sur l'extensibilité et la modularité du langage et de l'usine associée. Grâce à cette extensibilité, il est possible d'introduire la définition et l'interprétation de nouvelles constructions dans le langage FRACTAL ADL (répartition, trace, aspects, empaquetage, etc.). Cependant, le langage FRACTAL ADL ne définit aucune extension pour simplifier la définition de constructions récurrentes dans

les architectures et pour assurer des invariants architecturaux. De telles extensions permettent notamment de fiabiliser le développement des architectures à base de composants.

3.3.2 Modèle de composants OPENCOM

OPENCOM est un modèle de composants développé par l'université de Lancaster dans le cadre du projet OPENORB [BCA+01]. OPENCOM est un modèle de composants léger, efficace et réflexif adapté au développement des intergiciels [CBG+04]. Des implantations en C++ et en Java sont disponibles pour ce modèle de composants [Nex05]. OPENCOM empreinte à la technologie COM (*Component Object Model*)[1] ses propriétés d'interopérabilité au niveau binaire, de langage de description d'interfaces (IDL) (en anglais, *Interface Description Language*), des identifiants globalement uniques et l'interface de contrôle IUnknown pour supporter l'introspection du composant. Dans OPENCOM, les concepts de base reposent sur les notions d'*interface*, de *réceptacle* et de *connexion* comme illustré dans la figure 3.6. Une *interface* représente un service fourni tandis qu'un *réceptacle* décrit un service requis. La *connexion* correspond à une liaison entre un service requis et un service fourni de même type. OPENCOM permet de déployer un support d'exécution (OpenCOM runtime) par espace d'adressage afin de pouvoir créer/détruire/lier les composants qu'il héberge. Ce support d'exécution maintient un graphe des composants en cours d'exécution qu'il héberge (System graph). La compatibilité du support d'exécution avec les composants qu'il héberge repose sur l'implantation des interfaces *IMetaInterface*, *IConnections* et *ILifeCycle*. L'interface *IMetaInterface* fournit les opérations d'introspection du composant. L'interface *IConnections* contrôle la liaison des réceptacles du composant avec les interfaces. Enfin, l'interface *ILifeCycle* offre un support du cycle de vie des composants OPENCOM.

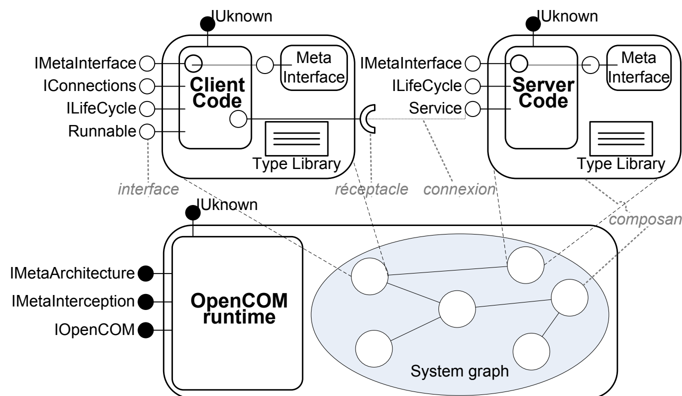


FIG. 3.6: Architecture à base de composants OPENCOM.

La contribution majeure d'OPENCOM repose sur la définition d'un méta-niveau supportant l'introspection et l'adaptation des composants. Ce méta-niveau est organisé de façon modulaire en différents méta-modèles. D'un point de vue **structurel**, deux méta-modèles sont disponibles : le méta-modèle IMetaInterface fournit une représentation locale d'un composant en terme d'interfaces requises et fournies tandis que le méta-modèle IMetaArchitecture fournit une représentation globale de liaisons et des contraintes fixées sur l'ensemble des composants. D'un point de vue **comportemental**, le méta-modèle IMetaInterception supporte la gestion de prétraitements et de post-traitements aux niveaux des interactions entre les composants.

Le langage de description et de reconfiguration d'architectures PLASTIK

Récemment, un langage de description et de reconfiguration d'architectures, appelé PLASTIK, a été défini pour faciliter la définition d'applications à base de composants OPENCOM [BJC05].

[1]Component Object Model Technologies : http://www.microsoft.com/com/

PLASTIK réutilise les constructions du langage de description d'architectures ACME [GMW00] et son extension *Armani* [Mon01] pour exprimer les invariants d'architecture. PLASTIK étend ACME/*Armani* pour introduire un support pour la reconfiguration dynamique d'architectures.

Le listing 3.3 présente les principaux concepts de ACME/*Armani* sur la description de l'architecture d'une pile de protocoles. La définition du style `PlastikMF` inclut deux ports (fourni et requis) et deux rôles (fourni et requis) (lignes 1–5). La définition du type de composant `OSIComp` constitue la notion centrale d'une pile de protocoles (lignes 6–10). La définition du type de connecteur `conn2Layers` permet l'assemblage des couches de la pile (lignes 11–14) tandis que l'invariant garantit qu'une pile de protocoles est obligatoirement composée de quatre couches (lignes 15–18).

```
1 Style PlastikMF {
2     Port Type ProvidedPort, RequiredPort;
3     Role Type ProvidedRole, RequiredRole;
4     ...
5 };
6 Component Type OSIComp : PlastikMF {
7     ProvidedPort Type upTo, downTo;
8     RequiredPort Type downFrom, upFrom;
9     Property Type layer = enum{application, transport, network, link};
10 };
11 Connector Type conn2Layers : PlastikMF {
12     ProvidedRole Type source;
13     RequiredRole Type sink;
14 };
15 Invariant
16     Forall c:OSIComp in sys.Components
17         cardinality(c.layer = application) = 1 and cardinality(c.layer = transport) = 1
18         and cardinality(c.layer = network) = 1 and cardinality(c.layer = link) = 1
```

LST. 3.2: Description ACME/*Armani* du type d'une pile de protocoles.

ACME/*Armani* ne supporte pas l'expression de la reconfiguration dynamique d'un système. Pour ce faire, le langage PLASTIK supporte les reconfigurations programmables et *ad-hoc* d'une application. Les reconfigurations programmables sont prévisibles lors de la conception de l'application. Par conséquent, PLASTIK permet au concepteur de spécifier l'action à appliquer lorsqu'une reconfiguration est nécessaire. PLASTIK supporte les reconfigurations *ad-hoc* des applications en fournissant un mécanisme de contrôle des modifications de l'architecture à l'exécution. Ainsi, si certaines reconfigurations ne peuvent être connues *a priori*, PLASTIK permet au concepteur de l'application de définir un certain nombre d'invariants d'architecture qui ne pourront pas être violés par les reconfigurations *a posteriori* de l'application.

Le listing 3.3 présente un exemple de définition d'une pile de protocoles à 4 couches `OSIStack` reconfigurable (ligne 1). Les composants et les connecteurs doivent être déclarés (lignes 2–8) avant d'être mis en relation (lignes 9–19). La dernière partie de la définition décrit les reconfigurations architecturales applicables à l'exécution (lignes 20–27). Dans cet exemple, le composant décodeur `MPEG-dec` est remplacé par un composant `H263-dec` lorsque la connexion au réseau devient insuffisante.

Cette description architecturale peut être déployée dans l'environnement d'exécution OPENCOM. Les reconfigurations architecturales et les invariants d'architectures sont alors garantis par un composant Configurator déployé par PLASTIK en même temps que l'application.

Bibliothèque de composants REMMOC

REMMOC est un canevas intergiciel de communication dédié au contexte des applications mobiles [Gra04, GBS05]. Ce contexte est caractérisé par une grande diversité des modes de communication (par exemple, RMI, *publish-subscribe*) et des protocoles de découverte (par exemple, UPNP, SLP) dont la connaissance est *a priori* inconnue lors du développement de l'application. Par conséquent, l'objectif de REMMOC est de fournir une plate-forme intergicielle capable de s'adapter automatiquement pour interagir avec les différents services disponibles dans un

```
1 System OSIStack : PlastikMF { // Application Level
2     Component MPEG-dec : OSIComp;
3     Component Transport : OSIComp;
4     Component Network : OSIComp;
5     Component Link : OSIComp;
6     Connector AppToTrans : conn2Layers;
7     Connector TransToNet : conn2Layers;
8     Connector NetToPhys : conn2Layers;
9     Attachments {
10        // connecting the Application to Transport layer
11        Application.dataTo to AppToTrans.source;
12        Transport.dataFrom to AppToTrans.sink;
13        // connecting the Transport to Network layer
14        Transport.dataTo to TransToNet.source;
15        TransToNet.sink to dynamic Network.dataFrom;
16        // connecting the Network to Physical layer
17        dynamic Network.dataTo to NetToPhys.source;
18        Link.dataFrom to OuterApplication;
19    };
20    On (Link.net_bandwidth = low) do {
21        detach MPEG-dec.downTo to AppToTrans.source;
22        remove MPEG-dec;
23        Component H263-dec : decoder = new decoder extended with {
24            Property decoder-type = "H263";
25        };
26        Attachments
27            H263-dec.downTo to AppToTrans.source;
28 };   };
```

LST. 3.3: Description PLASTIK d'une pile de protocoles.

contexte donné. Le modèle de composants utilisé pour développer cette plate-forme est le modèle OPENCOM. L'utilisation d'OPENCOM permet de réifier l'architecture de la plate-forme sous la forme de composants afin d'isoler les fonctionnalités nécessitant d'être adaptées automatiquement. Ces fonctionnalités adaptables sont identifiées sous les termes de Binding Component Framework et Service Discovery Component Framework. Le composant Binding Component Framework est utilisé par la plate-forme pour interagir avec les services hétérogènes disponibles dans le contexte d'exécution. Ce composant contient l'architecture de différents protocoles de communication supportés par la plate-forme et il est capable de remplacer à l'exécution un mode de communication synchrone par un mode de communication asynchrone. Le composant Service Discovery Component Framework est utilisé pour rechercher les différents services disponibles dans un environnement donné. Pour ce faire, ce composant dispose de différentes configurations correspondant aux différents standards définis en matière de découverte de services. Ce composant est capable d'utiliser plusieurs protocoles de découverte de services simultanément pour lister les différents services mis à disposition de la plate-forme dans un environnement donné.

La figure 3.7 présente un exemple d'architecture auto-adaptable construite en utilisant les composants de la bibliothèque REMMOC. Les composants Binding Component Framework et Service Discovery Component Framework sont connectés au reste de l'architecture et les configurations disponibles sont modélisées dans des *capsules*. Les *capsules* sont définies par le modèle de composants OPENCOM et correspondent à des assemblages de composants réutilisables. La capsule détaillée dans la figure 3.7 correspond à l'implantation du protocole de découverte UPNP. La plate-forme REMMOC peut également adapter le contenu des capsules lorsque les modifications du contexte d'exécution sont légères afin de modifier, par exemple, la qualité de service du mode de communication.

Synthèse

Le modèle de composants OPENCOM est reconnu pour la construction des intergiciels fiables et efficaces. La notion de *méta-niveau* proposée par OPENCOM permet d'ajouter de façon transparente des capacités d'introspection et de reconfiguration dynamique à des applications à base de composants. Si la version 1 d'OPENCOM était très liée au système d'exploitation Win-

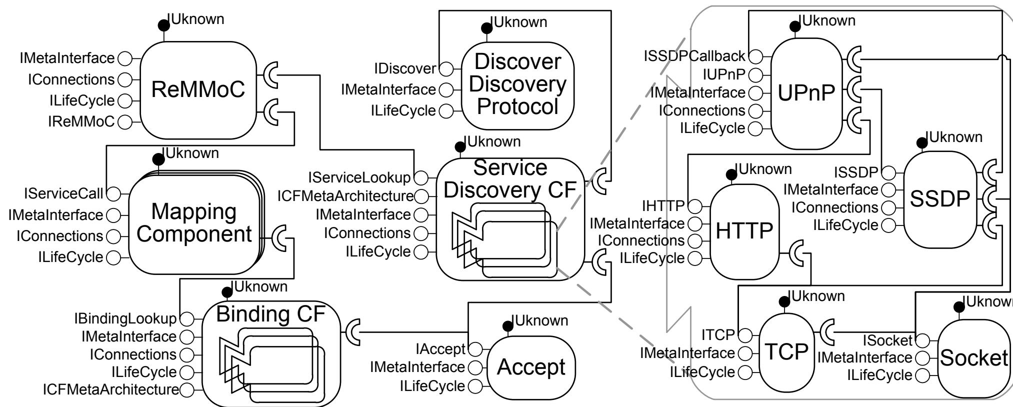


FIG. 3.7: Architecture à base de composants REMMOC.

dows (par son extension de la bibliothèque COM), la version 2 d'OPENCOM s'affranchit de cette dépendance et propose une version Java de son environnement d'exécution. Cependant, le développement d'un composant OPENCOM repose sur la réalisation d'un certain nombre d'interfaces imposées par le modèle de composants. Ces interfaces sont mélangées avec le code métier du composant ; elles fragilisent le développement des composants et introduisent une dépendance forte entre le code applicatif et le modèle de composants sous-jacent.

Notre intérêt pour le langage de description d'architectures PLASTIK porte sur la notion d'*invariant d'architecture* qui permet de fiabiliser la définition et la reconfiguration d'application à base de composants. Un invariant est une clause booléenne associée à un ensemble de composants de l'architecture et elle assure que les reconfigurations non-anticipées de l'application respectent les contraintes initialement définies par l'architecte. PLASTIK définit également un ensemble d'opérateurs de reconfiguration des architectures lorsque des modifications anticipées du contexte d'exécution sont détectées par l'environnement PLASTIK.

3.3.3 Modèle de composants ABC

Le modèle de composants ABC est une approche réflexive développée récemment par l'université de Pékin [Mei04, HMY06]. Ainsi, ABC permet de réifier un intergiciel existant sous la forme d'une architecture à base de composants en utilisant un mécanisme de réflexivité. L'utilisation de ce mécanisme n'impose aucune contrainte au développeur quant à la programmation de l'application et permet d'introduire les concepts de composants et d'architecture de manière transparente lors de l'exécution de l'application. Cette approche permet de contrôler de manière uniforme aussi bien des applications patrimoniales que des applications basées sur un standard de conception particulier (par exemple, J2EE [DK05], CCM [OMG02]).

L'objectif d'ABC est de définir une démarche de construction des applications centrée sur l'architecture. Ainsi, ABC propose de raisonner sur la notion d'architecture à toutes les étapes du cycle de vie de l'application comme illustré dans la figure 3.8.

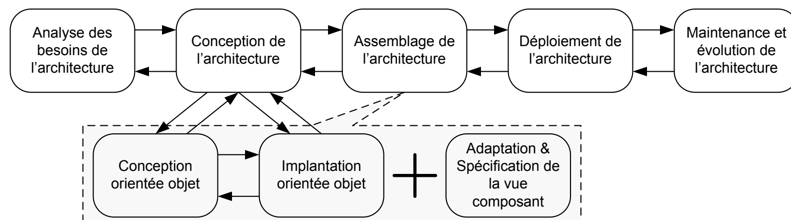


FIG. 3.8: Approche ABC centrée sur l'architecture.

Durant la phase d'analyse des besoins de l'architecture, les besoins du système à développer sont exprimés en terme de spécification de composants, de spécification de connecteurs et de spécifications de contraintes sans toutefois exprimer l'architecture du système à développer. Cette architecture est décrite lors de la phase de conception de l'architecture après transformation des différentes spécifications décrites lors de la phase précédente. Il peut alors être nécessaire d'adapter les différentes spécifications afin de répondre aux contraintes du système cible. Durant la phase d'assemblage de l'architecture, les composants, connecteurs et contraintes issus de la phase précédente sont sélectionnés et adaptés pour réaliser la logique de l'application. Les éléments qui ne peuvent pas être adaptés automatiquement sont réalisés manuellement en utilisant par exemple un langage objet. Ils sont ensuite testés et confrontés aux spécifications afin de respecter l'architecture. Lors de la phase de déploiement de l'architecture, la description de l'architecture est enrichie avec les contraintes de ressources et/ou de sécurité liée à l'utilisation des composants de l'application afin de faciliter le déploiement de l'application sur la plate-forme intergicielle. Enfin, lors de la phase de maintenance et évolution de l'architecture, la plate-forme intergicielle doit collecter les informations relatives à l'architecture afin de supporter l'évolution de l'application à l'exécution. Ces informations, généralement invisibles, nécessitent un mécanisme d'abstraction de haut niveau pour observer et manipuler l'architecture.

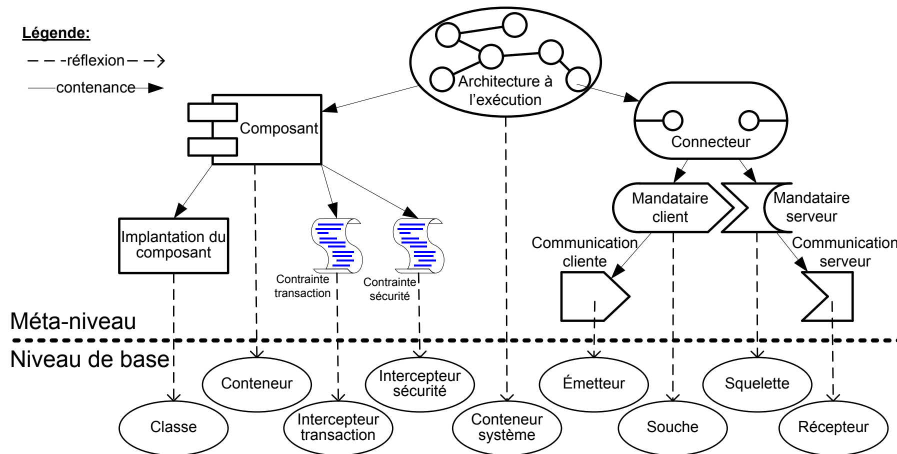


FIG. 3.9: Architecture à base de composants ABC.

Le méta-modèle de composants utilisé par ABC pendant l'exécution est illustré dans la figure 3.9. Dans ABC, une architecture à l'exécution est composée d'un ensemble de composants et de connecteurs. Chaque composant comporte une implantation et différentes contraintes techniques (transactions, sécurité, etc.). Les connecteurs sont décomposés en deux mandataires réalisant les communications en utilisant un mécanisme de communication. Chacun de ces éléments du méta-modèle ABC est associé à une entité de l'environnement d'exécution. Ainsi, une classe de contenu est associée à l'implantation du composant, le conteneur de la classe correspond au composant et les intercepteurs positionnés sur le conteneur correspondent aux contraintes techniques du composant.

Langage de description d'architecture pivot ABC/ADL

Dans la continuité de sa démarche centrée sur l'architecture, ABC définit un langage de description d'architectures de haut niveau [MCWF02]. Le langage ABC/ADL s'inspire des langages de description d'architecture tels que Wright [All97], Unicon [SDK+95], et ACME [GMW00] pour définir sa syntaxe.

L'objectif d'ABC/ADL est non seulement de fournir un langage de description de haut niveau pour décrire des architectures logicielles mais aussi de fournir un mécanisme de projection

automatique vers des architectures logicielles concrètes via un mécanisme de règles de transformation et de connecteurs configurables. Les principes appliqués par ABC/ADL visent à fournir un langage simple et intelligible de description d'architecture tout en autorisant différentes extensions. La structure du langage ABC/ADL repose sur trois couches. La couche méta fournit les constructions abstraites pour définir les styles architecturaux et les patrons. La couche definition fournit le langage concret permettant de décrire les composants, les connecteurs et l'architecture de l'application. Enfin, la couche instance fournit les abstractions nécessaires pour exprimer les liaisons entre les composants et les connecteurs.

Le langage ABC/ADL différencie les notions de type et d'instance non seulement au niveau des composants mais aussi au niveau des interfaces afin de séparer les méthodes généralement associées à un type de composant (par exemple, création, recherche de composants) des méthodes associées au métier du composant. Le langage ABC/ADL supporte la définition hiérarchique de composants afin de mieux contrôler le processus de conception de l'application.

Le listing 3.4 présente un exemple de description d'un composant `DatingManager` s'appuyant sur le style architectural `BLACKBOARD.BlackBoard` et d'un connecteur J2EE `J2EEConnector`. Le composant `DatingManager` (lignes 1–12) fournit une méthode de type `findByPrimaryKey()` permettant de retrouver les instances de ce composant en fonction de leur clé (ligne 4). Dans la définition du connecteur `J2EEConnector` (lignes 13–21), l'utilisation du mot-clé * (joker) (lignes 15–16) spécifie que la liste des méthodes du connecteur correspond à la liste des méthodes fournies et requises par les composants qui lui sont connectés.

```
1  Component DatingManager is BLACKBOARD.BlackBoard {
2      Interfaces {
3          provide player DatingManager is BlackBoard.Entry {
4              type-method { DatingManager findByPrimaryKey(Object id); }
5              instance-method { ... }
6          }
7          request player Agenda is BlackBoard.Notification {
8              type-method { ... }
9              instance-method { ... }
10     }   }
11     ...
12 }
13 Connector J2EEConnector is DEFAULT.Connector {
14     Interfaces {
15         provide player Callee is Connector.Callee{*}
16         request player Caller is Connector.Caller{*}
17     }
18     Properties{ Platform = J2EE; }
19     Dependencies{ Callee depends on Caller; }
20     SemanticDescription { Caller includes Callee; }
21 }
```

LST. 3.4: Description ABC/ADL d'un composant et d'un connecteur.

La définition d'une architecture avec ABC/ADL est illustrée dans le listing 3.5 et comporte deux parties. La partie délimitée par le mot-clé `uses` correspond à la déclaration des composants et des connecteurs utilisés dans l'architecture (lignes 2–10). La partie délimitée par le mot-clé `Config` correspond à la configuration des composants de l'architecture (lignes 11–18). L'utilisation d'une variable `i` (ligne 9) permet de simplifier les connexions entre les composants et les connecteurs de l'architecture. ABC/ADL supporte la définition de plusieurs configurations que l'utilisateur peut choisir *a posteriori*. L'architecture ainsi obtenue peut être attachée à la définition d'un composant composite en utilisant le mot-clé `Structure` et en connectant les interfaces du composite à l'architecture via le mot-clé `mapping`.

Cette architecture abstraite d'une application métier peut ensuite être projetée vers une plate-forme intergicielle particulière telle que J2EE en spécifiant la conversion des concepts d'ABC/ADL vers J2EE. Cette conversion spécifie par exemple que les méthodes de type du composant correspondent à la notion de maison de composants dans J2EE. Les éléments qui ne peuvent pas être transformés automatiquement sont implantés manuellement et viennent compléter l'architecture décrite avec ABC/ADL.

```
1  Architecture DS_Architecture {
2      uses {
3          Component agendas : Agenda[];
4          Component datingManager : DatingManager;
5          Component ruleManager : RuleManager;
6          Connector agendaToDatingManager : SecuredConnector[];
7          Connector agendaToRuleManager : DefaultConnector[];
8          Connector datingManagerToAgenda : DefaultConnector[];
9          Variable i : int;
10     }
11     Config main {
12         agendas[i].DatingManager connects agendaToDatingManager[i].Callee
13         agendaToDatingManager[i].Caller connects datingManager.DatingManager
14         agendas[i].RuleManager connects agendaToRuleManager[i].Callee
15         agendaToRuleManager[i].Caller connects ruleManager.RuleManager
16         datingManager.Agenda connects datingManagerToAgenda[i].Callee
17         datingManagerToAgenda[i].Caller connects agendas[i].Agenda
18     }
19     SemanticDescription { }
20 }
21 Component Dating_System is System{
22     Structure { architecture DS_Architecture }
23     mapping { self.makeMeeting to datingManager.makeMeeting }
24 }
```

LST. 3.5: Description ABC/ADL d'un composant et d'un connecteur.

Serveur d'applications PKUAS

Le serveur d'applications PKUAS [MH04] constitue un exemple d'utilisation de l'approche ABC pour construire des intergiciels adaptables. Cette plate-forme intergicielle compatible avec le standard J2EE [DK05] utilise ABC pour fournir automatiquement une représentation intelligible sous forme de composants de sa structure et des applications qu'elle héberge.

Pour ce faire, PKUAS définit un méta-modèle composé de deux couches pour représenter une application J2EE que nous illustrons dans la figure 3.10. La couche supérieure utilise les concepts généraux application software architecture, component, constraint, et connector qui sont définis par le modèle ABC. La couche inférieure réalise la conversion entre les concepts ABC et les concepts J2EE en fournissant une représentation de l'application basée sur la technologie JMX [Sun02]. Les opérations mises à disposition par ces méta-objets permettent cinq types de manipulations :

1. La gestion du cycle de vie des éléments de base associés aux méta-objets,
2. L'ajout/suppression/remplacement des services de la plate-forme PKUAS (à l'exception du service de communication),
3. La gestion des statistiques disponibles sur l'état interne et le comportement des objets de base,
4. L'invocation des méthodes métiers des objets de base par réflexion,
5. L'observation et la modification des états et comportements des objets de base par réflexion.

Les évaluations de performances réalisées sur PKUAS montrent que le surcoût de l'utilisation d'un mécanisme de réflexion pour maintenir un lien de causalité entre l'architecture d'une application et sa structure en cours d'exécution est acceptable [MH04, HMY06].

Synthèse

Le modèle de composants ABC propose une solution transparente pour réifier à l'exécution une application patrimoniale sous la forme d'une architecture à base de composants. Pour ce faire, ABC définit un méta-modèle minimal composé des notions de composant, connecteur et contrainte. Ce méta-modèle peut être raffiné en différents méta-modèles adaptés à des plates-formes d'exécution particulières (CCM, J2EE, Fractal, etc.) qui seront en mesure de définir les correspondances entre les artefacts exécutables de l'application et les éléments du méta-modèle.

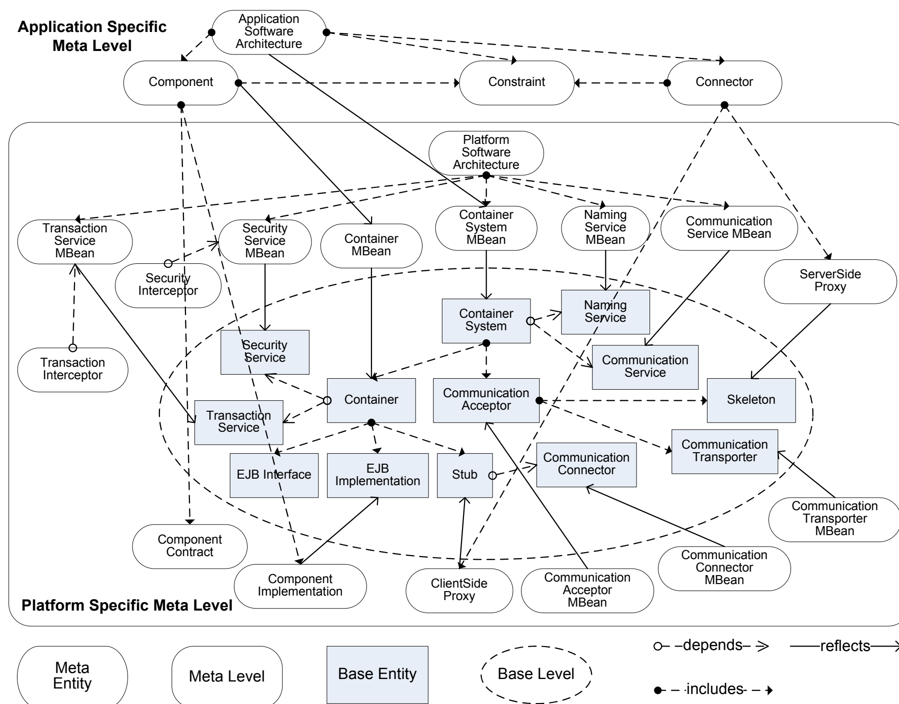


FIG. 3.10: Présentation du méta-modèle à 2 couches de PKUAS.

De plus, la définition de l'architecture en utilisant le langage ABC/ADL permet d'assurer une interopérabilité du code source de l'application en fournissant des règles de transformation du méta-modèle minimal vers une plate-forme cible déterminée.

Notre intérêt pour le langage de description d'architectures ABC/ADL porte non seulement sur la généricité du langage mais aussi sur la notion de *variable* permettant de systématiser les constructions répétitives d'une architecture à base de composants. Cependant, l'absence d'un modèle de programmation limite la fiabilité des applications en réduisant les possibilités de vérification du code applicatif vis-à-vis des contraintes liées au modèle de composants ABC.

3.3.4 Autres approches

Cette section ne présente pas une démarche de construction complète à base de composants, mais elle s'intéresse de façon spécifique au modèle de programmation des composants ARCHJAVA et au langage de description d'architectures DARWIN pour le support des constructions récurrentes.

Modèle de composants ARCHJAVA

Le modèle de composants ARCHJAVA est une extension du langage Java permettant de structurer un code applicatif en utilisant les notions de composants, ports et connecteurs [ACN02a, ACN02b]. Contrairement aux modèles de composants présentés précédemment, ARCHJAVA intègre les différents éléments de description d'architecture au sein du code des composants afin de garantir que les spécifications architecturales sont respectées par l'implantation des composants.

Le listing 3.6 présente l'implantation d'un composant `Parser` (ligne 1) définissant deux ports `in` (lignes 2–5) et `out` (lignes 6–9). Chaque port définit un ensemble de signatures de méthodes pouvant être fournies, requises ou diffusantes par le composant. Le code des méthodes associées aux ports du composant est ensuite défini (lignes 10–17).

```
1 public component class Parser {
2     public port in {
3         provides void setInfo(Token symbol, SymTabEntry e);
4         requires Token nextToken() throws ScanException;
5     }
6     public port out {
7         provides SymTabEntry getInfo(Token t);
8         requires void compile(AST ast);
9     }
10    void parse(String file) {
11        Token tok = in.nextToken();
12        AST ast = parseFile(tok);
13        out.compile(ast);
14    }
15    AST parseFile(Token lookahead) { ... }
16    void setInfo(Token symbol, SymTabEntry e) { ... }
17    SymTabEntry getInfo(Token t) { ... }
18    ...
19 }
```

LST. 3.6: Description ARCHJAVA d'un composant primitif analyseur lexical.

Le listing 3.7 décrit un composant `Compiler` (ligne 1) réalisé par un assemblage de trois composants primitifs identifiés par le mot-clé **`final`** (lignes 2–4). Ces composants sont connectés entre eux (lignes 5–6) et le code du composant restant (lignes 7–12) détermine le comportement du composant lorsque l'application est démarrée.

```
1 public component class Compiler {
2     private final Scanner scanner = ... ;
3     private final Parser parser = ... ;
4     private final CodeGen codegen = ... ;
5     connect scanner.out, parser.in;
6     connect parser.out, codegen.in;
7     public static void main(String args[]) {
8         new Compiler.compile(args);
9     }
10    public void compile(String args[]) {
11        // for each file in args do:
12    ... parser.parse(file); ...
13 }  }
```

LST. 3.7: Description ARCHJAVA de l'assemblage d'un compilateur.

Synthèse

Le modèle de composants ARCHJAVA supporte la construction d'applications Java à base de composants. Ainsi, le modèle de programmation ARCHJAVA étend la grammaire du langage Java afin d'y introduire les notions dédiées à la programmation par composants. Cette approche facilite la mise en œuvre des composants applicatifs en offrant un langage sémantiquement plus riche que les langages orientés objets. Cette extension permet également d'assurer les règles de programmation imposées par le modèle de composants, telle que l'*intégrité des communications* via l'utilisation du mot-clé `connect`. Cependant, le principal défaut d'ARCHJAVA réside dans le mélange de la description d'architecture avec la programmation des composants. Ce mélange nuit à la séparation des préoccupations mises en jeu dans la construction d'un intergiciel. En effet, la définition d'une application avec ARCHJAVA devient alors plus complexe car elle nécessite de prendre en compte des aspects architecturaux et algorithmiques dans un même contexte. De plus, le modèle de composants ARCHJAVA ne supporte ni le partage de composants ni le contrôle d'exécution.

Langage de description d'architectures DARWIN

Le langage de description d'architectures DARWIN supporte la notion de composant comme entité de base de la description d'une architecture [MDEK95]. DARWIN distingue les composants primitifs (encapsulant du code fonctionnel) des composants composites (encapsulant d'autres composants) afin d'organiser de façon hiérarchique la structuration des applications.

La plate-forme intergicielle REGIS fournit un support d'exécution en C++ pour les composants décrits en utilisant le langage de description DARWIN [MDK94]. Cette plate-forme définit trois états pour les composants DARWIN : *actif*, *passif* ou *gelé*. L'état *actif* correspond à un composant en cours d'exécution pouvant initier et accepter des requêtes. L'état *passif* correspond à un composant en cours d'exécution pouvant accepter des requêtes mais ne pouvant pas en initier. Enfin, l'état *gelé* correspond à un composant n'étant ni à l'origine, ni en cours de traitement d'une requête. La définition de ces états permet d'appliquer différentes opérations de reconfiguration telle que l'*ajout d'un composant*, la *modification* d'une liaison lorsque le composant client est gelé ou le *retrait d'un composant* lorsque celui-ci est gelé et que toutes ses liaisons ont été supprimées.

Le listing 3.8 présente la description DARWIN d'une chaîne de délégation générique écrite sous la forme d'un assemblage de composants paramétrable. En effet, la description du composant `Pipeline` est configurable par un attribut entier `n` (ligne 5) représentant le nombre de chaînons de type `Link` composant la chaîne de délégation. Chaque chaînon fournit une interface `input` et requiert une interface `output` (lignes 1–4). Le composant `Pipeline` déclare, dans un premier temps, une liste bornée de chaînons (ligne 8) puis il procède à la configuration des maillons en les instanciant (ligne 10) et en les liant à leur suivant (ligne 12). Lors de l'instanciation, il est possible de spécifier l'emplacement d'un composant en utilisant l'opérateur `@` suivi de sa position (ligne 10). Pour faciliter les configurations répétitives, le langage de description DARWIN fournit également un opérateur de boucle `forall` permettant de factoriser certaines parties de la description d'architecture (ligne 9). Le mot-clé `when` permet de réaliser un contrôle conditionnel d'application de la description (ligne 11).

```
 1 component Link {                  // déclaration d'un chaînon
 2     provide input;
 3     require output;
 4 }
 5 component Pipeline (int n) {
 6     provide input;
 7     require output;
 8     array : elt[n] : Link;        // déclaration d'une liste de chaînons
 9     forall k : 0..n-1 {
10         inst elt[k] @ k+1;        // instanciation d'un chaînon
11         when k < n;               // liaison des chaînons entre eux
12             bind elt[k+1].input -- elt[k].output;
13     }
14     bind
15         elt[0].input -- input;    // exportation de l'interface fournie du premier chaînon
16         output -- elt[n-1].output; // importation de l'interface requise du dernier chaînon
17 }
```

LST. 3.8: Description DARWIN d'une chaîne de délégation.

Synthèse

Le langage de description d'architectures DARWIN permet de décrire une application sous la forme d'un assemblage hiérarchique de composants paramétrables. Notre intérêt pour le langage DARWIN porte sur le fait qu'il définit des opérateurs de factorisation des constructions d'une architecture. Ainsi, la notion de collection et d'opérateur de boucle permet de systématiser des opérations répétitives, verbeuses et potentiellement sujettes à de nombreuses erreurs de l'architecte. Cependant, la dépendance de DARWIN vis-à-vis de la plate-forme d'exécution REGIS offre peu de solutions en terme d'extensibilité du langage et de l'outillage sous-jacent.

3.4 Conclusion

Ce chapitre a présenté un état de l'art centré sur les approches existantes en matière de développement fiable d'intergiciels adaptables. Nous avons introduit dans un premier temps les concepts fondamentaux relatifs à la programmation par composants, aux architectures logicielles et aux canevas intergiciels. Ces concepts se sont imposés dans le domaine de la construction des intergiciels et permettent de comprendre, fiabiliser, faciliter et pérenniser la définition d'une architecture logicielle.

Ensuite, nous avons illustré l'application de ces concepts sur trois modèles de composants que nous avons jugé compatible avec notre approche. En effet, les modèles de composants industriels tels que EJB, CCM ou SCA correspondent à des modèles lourds de par leur difficulté de mise en œuvre et leur coût à l'exécution. Par conséquent, nous nous sommes intéressés à des approches plus légères dont l'application au développement des intergiciels a pu être validée par le passé.

Une synthèse des caractéristiques des modèles de composants est présentée dans le tableau 3.2. Cette synthèse compare les modèles de composants et les langages de descriptions de différentes approches de conception orientées composants. Les critères évalués pour les **modèles de composants** concernent le support du *partage* des composants, l'*extensibilité* du modèle, c.-à-d. les niveaux de contrôle et le modèle de *programmation*. Les critères observés entre les **langages de description d'architecture** concernent la *factorisation* des construction récurrentes, la *vérification* des architectures et l'*extensibilité* du langage de description et de son interpréteur.

De cette étude, il apparaît que le modèle de composants ARCHJAVA possède un modèle de programmation fiable et riche pour la programmation des composants primitifs d'un système. Ainsi, son extension du langage Java permet de réduire la taille du code des composants primitifs comparé aux approches mises en œuvre dans les modèles FRACTAL et OPENCOM. L'utilisation d'un modèle de programmation contraint permet également d'assurer des propriétés sur l'architecture des composants telles que l'*intégrité des communications* là où l'approche réflexive d'ABC est incapable de vérifier ce type de propriété.

Par contre, le langage de description d'architectures d'ARCHJAVA présente le défaut d'être regroupé avec le modèle de programmation. Cet inconvénient rend plus complexe la séparation des préoccupations et impacte par conséquent le passage à l'échelle des applications construites avec le modèle de composants ARCHJAVA. Sur cet aspect de passage à l'échelle, les langages de description d'architecture DARWIN et ABC/ADL propose des opérateurs de factorisation des constructions récurrentes dans les architectures logicielles. Ces constructions permettent non seulement de simplifier les descriptions d'architectures mais elles permettent également de réduire le nombre d'erreurs pouvant apparaître lors de la description manuelle d'architectures de grande envergure. Concernant l'évolution dynamique d'une architecture à base de composants, seul le langage de description PLASTIK propose un formalisme pour supporter des évolutions anticipée ou non de l'architecture à l'exécution. En particulier, PLASTIK définit la notion d'invariant d'architecture pour vérifier que toute reconfiguration non anticipée de l'architecture ne laisse pas l'application dans un état incohérent.

À la vue des avantages et inconvénients des différentes approches que nous avons étudié dans ce chapitre, nous choisissons d'utiliser le modèle de composants FRACTAL pour développer notre proposition GOTM. Ce choix est motivé par le support des *architectures hiérarchiques avec partage* et la *flexibilité du modèle de composants et de son outillage*. En particulier, nous comptons nous appuyer sur cette flexibilité pour contribuer à une amélioration du modèle de programmation de FRACTAL et du langage de description d'architectures FRACTAL ADL. Notre contribution au modèle de programmation FRACTAL vise à enrichir le langage de programmation avec une sémantique riche et adaptée à la programmation des composants afin de simplifier et fiabiliser la tâche du développeur de composants (comme le propose ARCHJAVA). Notre objectif est également de réduire la dépendance des composants d'une application avec le modèle de composants les réalisant (comme le propose ABC). Notre contribution au langage de description FRACTAL ADL adresse la factorisation et la vérification des motifs présents dans les architectures logicielles. Notre objectif est non seulement de faciliter la description d'architectures complexes (comme le proposent DARWIN ou ABC/ADL) mais aussi de garantir un certain nombre de pro-

Critères	Modèle de composants			Langage de description d'architectures		
	partage	*extensibilité*	*programmation*	*factorisation*	*vérification*	*extensibilité*
FRACTAL/FRACTAL ADL	**OUI**	**FORTE**	LOURD	NON	FAIBLE	**FORTE**
OPENCOM/PLASTIK	NON	**FORTE**	LOURD	**OUI**	**FORTE**	FAIBLE
ABC/ABC/ADL	NON	**FORTE**	**LÉGER**	**OUI**	FAIBLE	**FORTE**
ARCHJAVA	NON	FAIBLE	**LÉGER**	**OUI**	**FORTE**	FAIBLE
REGIS/DARWIN	NON	FAIBLE	N/A	**OUI**	**FORTE**	FAIBLE
Approche GOTM	**OUI**	**FORTE**	**LÉGER**	**OUI**	**FORTE**	**FORTE**

TAB. 3.2: Synthèse des modèles de composants.

priétés architecturales lors de l'exécution de l'application (comme le propose PLASTIK).

Deuxième partie

Contributions

Investment in software reliability will increase until it exceeds the probable cost of errors.

Murphy's Law

Chapitre 4

La démarche GoTM de construction de services de transactions hautement adaptables

Sommaire

4.1 Introduction **62**

4.2 Motivations **62**

4.2.1 Limitations des canevas logiciels transactionnels actuels 63

4.2.2 Objectifs d'un canevas de services de transaction hautement adaptables . 63

4.2.3 Démarches de construction envisageables 64

4.2.4 Approche de construction à très fine granularité 65

4.3 Vers des composants de très fine granularité **66**

4.3.1 Principes de conception des composants de très fine granularité 66

4.3.2 Illustration de notre démarche de construction 67

4.4 Présentation du canevas logiciel GoTM **70**

4.4.1 Démarche de construction de GoTM 70

4.4.2 Modèle de programmation des composants 70

4.4.3 Langage de description et de vérification de motifs d'architecture 71

4.4.4 Bibliothèque de composants transactionnels GoTM 72

4.4.5 Modèles dédiés à la configuration des services de transactions 72

4.5 Organisation de la deuxième partie **72**

*C*E CHAPITRE PRÉSENTE LA DÉMARCHE DE CONSTRUCTION *que nous avons appliquée dans le cadre de cette thèse pour construire des services de transactions hautement adaptables. Cette démarche repose sur la définition d'un canevas logiciel à base de composants de granularité très fine. Grâce à ce degré de granularité, il est possible d'adapter différentes caractéristiques du service de transactions (modèle de transactions, protocole de validation, standard transactionnel, etc.). Cependant, ce choix de conception nécessite également de définir un outillage adapté afin de simplifier et de fiabiliser le développement du canevas logiciel face à l'augmentation des composants et à la complexification des architectures.*
Nous proposons donc d'appliquer la philosophie des exogiciels *en complétant cette dernière avec quatre contributions : un modèle de programmation des composants, un langage de description et de vérification de motifs d'architecture, une bibliothèque de composants dédiés au domaine du transactionnel et des modèles de haut niveau pour la configuration des services de transactions. Nous montrons que l'application de cette démarche nous permet de construire des services de transactions hautement adaptables dont les performances sont comparables aux services disponibles sur le marché.*

4.1 Introduction

Les canevas logiciels existants adressent l'adaptabilité sous diverses manières. C'est ainsi que la programmation par aspects [LPP+05], la programmation réflexive [Bar98], la programmation générative [Bri05] et la programmation par composants [Par05] ont été employées à plusieurs reprises pour définir des canevas logiciels adaptables tels que *Java Aspect Components* [PSD+04], *Reflective Transaction Framework* [BP97], ou OPENORB [BCA+01]. Toutes ces approches ont démontré qu'il est possible de construire des intergiciels adaptables en utilisant des paradigmes de conception évolués (composants, modèles, aspects, etc.). Cependant, la granularité employée dans ces approches limite l'adaptabilité des intergiciels construits à un nombre prédéfini de caractéristiques.

Dans le cadre de cette thèse, nous souhaitons définir un canevas logiciel dédié à la construction de services de transactions hautement adaptables. Pour ce faire, nous nous inspirons de la philosophie des *exogiciels* [Qué05] et nous proposons de compléter cette démarche afin d'en accroître les propriétés d'adaptabilité. Ainsi, nous définissons un modèle de programmation fiable pour le modèle de composants généraliste. Le langage de description d'architecture est également étendu pour intégrer un langage de description et de vérification de motifs d'architecture. Dans le cadre de notre étude du domaine transactionnel, nous avons développé une bibliothèque de composants communs à différents services de transactions. Ce développement s'appuie sur nos deux premières contributions et propose de réifier les éléments caractéristiques d'un service de transactions sous forme de motifs de conception. Enfin, nous proposons un ensemble de modèles de haut niveau dédiés à la configuration des services de transactions construits à partir de notre bibliothèque de composants.

La suite de ce chapitre est organisée de la manière suivante. La section 4.2 présente les motivations du canevas logiciel GOTM. La section 4.3 illustre les enjeux de la construction de canevas logiciel à très fine granularité. La section 4.4 présente les différentes contributions de GOTM. Enfin, la section 4.5 présente l'organisation de la suite de cette partie.

4.2 Motivations

Cette section expose les limitations des canevas logiciels dédiés au domaine du transactionnel. Afin de faire face à ces limitations, nous proposons ensuite d'employer une démarche de construction à granularité extrêmement fine. Grâce à cette démarche, il est possible d'accroître les capacités d'adaptation structurelle et comportementale du canevas logiciel.

4.2.1 Limitations des canevas logiciels transactionnels actuels

L'**adaptabilité structurelle** des services de transactions a été étudiée à plusieurs reprises afin de construire des services de transactions supportant différents modèles de transactions étendus. La définition du canevas ACTA [CR90] a permis de formaliser la définition des modèles de transactions. Il est ainsi possible de définir des modèles de transactions tels que les SAGAS [GMS87] ou les transactions imbriquées [Nem04] en utilisant la syntaxe définie par ACTA. Cette formalisation basée sur un langage unique a donné lieu à différentes études sur la définition d'un canevas logiciel issu de ACTA commun à plusieurs modèles de transactions. Le service de transactions réflexif défini par R. Barga [Bar98] propose de construire un service de transactions utilisant un modèle de transactions décrit par l'application (voir section 2.3.2 du chapitre 2). Le service de transactions ATPMos [Fab05] isole ces modèles de transactions et propose une définition du service de transactions étendues comme un ensemble de briques logicielles élémentaires et combinables inspirées du canevas ACTA (voir section 2.3.3 du chapitre 2). Ces briques sont ensuite combinées en fonction des besoins de l'application pour construire une instance de service particulière. Cependant, la granularité de ces travaux ne supporte l'adaptation que d'une seule dimension d'un service de transactions : le modèle de transactions. Par conséquent, ces services de transactions ne sont qu'éventuellement basés sur un standard transactionnel reconnu et ne maîtrisent pas d'autres aspects tels que la configuration du protocole de validation.

L'**adaptabilité comportementale** des services de transactions telle qu'elle est adressée actuellement se résume à la mise en place d'un mécanisme de courtage dédié aux services de transactions [Hér05, AK05]. Ces approches se contentent de lister et caractériser les différentes particularités des services de transactions (comme vu dans les sections 2.3.4 et 2.3.5 du chapitre 2). Ces caractéritistiques peuvent être concrètes (modèle de transaction, standard transactionnel, etc.) [Hér05] ou abstraites (atomicité, consistence, etc.) [AK05]. Le service est ensuite sélectionné par l'application en fonction des caractéristiques transactionnelles requises. Cependant, la combinaison des différentes possibilités des caractéristiques résulte en une explosion combinatoire du nombre de services de transactions devant être mis à disposition de l'application. Cette approche n'est pas raisonnable à la vue du nombre de services de transactions disponibles sur le marché. La granularité employée ici, c.-à-d. le service de transactions, requiert de développer plusieurs services de transactions chaque fois qu'une caractéristique évolue.

Enfin, la **démarche de construction** des services de transactions a rarement été abordée dans les travaux existants. Étant donné la granularité adoptée dans ces travaux, de nombreuses approches se contentent de réutiliser des services de transactions patrimoniaux pour les adapter à leurs besoins [Bar98, Nem04]. Les éléments caractéristiques d'un service de transactions sont donc rarement capitalisés par les approches existantes (à l'exception d'ArjunaTS comme présenté dans la section 2.3.6 du chapitre 2). Le manque de structuration interne du service de transactions limite la vision architecturale que peuvent offrir ces types de services de transactions. Nous souhaitons donc mettre l'accent sur l'ouverture de l'architecture du service en offrant une vision très fine de l'architecture des services de transactions que nous sommes en mesure de construire. Cette approche très fine contribue à faciliter tant l'adaptation structurelle que l'adaptation comportementale de nos services de transactions.

4.2.2 Objectifs d'un canevas de services de transaction hautement adaptables

Notre objectif est de développer des services de transactions hautement adaptables. Cette haute adaptabilité adresse non seulement les aspects structurels et comportementaux de l'adaptabilité mais aussi la granularité de l'adaptation. Comme nous avons pu le constater dans le chapitre 2, il existe deux types d'approches. Le premier type d'approches se limite à l'adaptation structurelle des modèles de transactions ou à l'adaptation comportementale des services de transactions. Le second type d'approches se limite à l'intégration des transactions des services de transactions dans les plates-formes applicatives ou les applications.

Notre proposition, illustrée dans la figure 4.1, adresse les différents aspects des deux types d'approches : l'adaptation et l'intégration. De plus, nous proposons d'adresser l'adaptation des

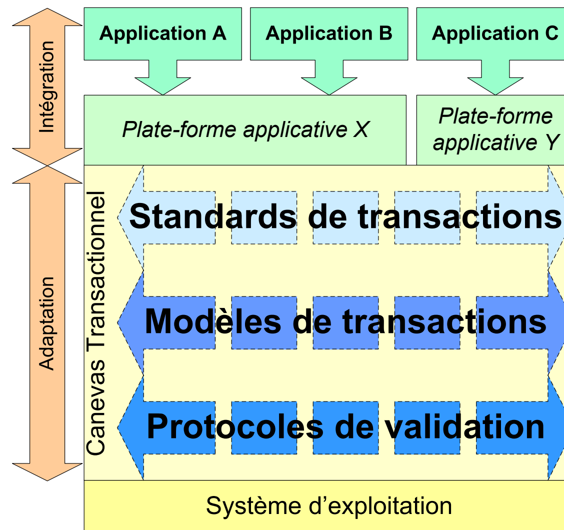


FIG. 4.1: Illustration des objectifs d'un service de transactions hautement adaptable.

services de transactions à une granularité beaucoup plus fine que les approches actuelles. Ainsi, nous souhaitons non seulement adapter le **service de transactions** ou le **modèle de transactions**, mais aussi des aspects plus originaux tels que les **standards de transactions** ou les **protocoles de validation**.

Cependant, cette granularité d'adaptation requiert de modifier l'approche traditionnelle de conception des services de transactions afin de supporter un paradigme de programmation beaucoup plus modulaire. Par conséquent, nous proposons une nouvelle approche de construction à très fine granularité des canevas intergiciels s'appuyant sur les paradigmes de programmation par composants et d'architecture logicielle.

4.2.3 Démarches de construction envisageables

Les démarches envisageables pour réaliser un service de transactions hautement adaptable peuvent prendre différentes formes. Notamment, les approches issues de l'ingénierie des modèles, des langages dédiées et des canevas à base de composants sont des candidates potentielles pour atteindre notre objectif.

Ingénierie des modèles. Dans cette approche, le domaine cible, c.-à-d. les services de transactions, est capturé par un méta-modèle décrivant exhaustivement les caractéristiques du domaine et ses points de variation. La description d'un service de transactions réalisée à partir de ce méta-modèle peut être utilisée pour générer un instance de service de transactions fidèle à cette description. Cependant, l'implantation concrète de ce service et les transformations propres à la génération de l'instance du service hautement adaptable doivent également être modélisés sous une forme particulière.

Langages dédiés. Dans cette approche, l'objectif est de définir un langage de programmation du service de transactions hautement adaptable en utilisant des abstractions de plus haut niveau que les langages de programmation génériques. Dans ce cas, les caractéristiques du domaine sont réifiées sous la forme de constructions dédiées dans le langage. Le programme décrit à l'aide de ce langage est ensuite compilé pour obtenir un instance de service de transaction hautement adaptable. De nouveau, il est nécessaire de connaître la forme finale du service de transactions hautement adaptable pour définir le processus de compilation du langage dédié vers l'instance du service.

Canevas à base de composants. Cette approche se positionne à un niveau d'abstraction moindre puisqu'elle consiste à réifier les caractéristiques du domaine sous la forme de composants intergiciels. La composition de ces composants dédiés permet d'obtenir une instance de service de transactions. Par conséquent, les composants répondent au problème de la modélisation du substrat d'exécution mais ils n'offrent que peu de d'abstraction de la conception d'un service de transaction hautement adaptable.

Convergence des approches. Notre approche vise donc à tirer parti des avantages des différentes approches existantes. Nous souhaitons ainsi utiliser le paradigme de composant pour réaliser le service de transactions hautement adaptable. Pour développer les composant de ce service, nous souhaitons définir un langage dédié fournissant une abstraction suffisante pour faciliter et fiabiliser la réalisation des composants. Enfin, pour modéliser le service de transactions à réaliser, nous souhaitons utiliser un méta-modèle de haut niveau d'abstraction afin de contrôler la définition du service de transactions hautement adaptable.

4.2.4 Approche de construction à très fine granularité

Pour développer des services de transactions fournissant une vision architecturale très fine, nous proposons de définir un canevas logiciel à base de composants de très fine granularité. Notre objectif est d'identifier les fonctionnalités élémentaires mises en jeu dans un service de transactions et de réifier celles-ci sous forme de composants. En utilisant une granularité beaucoup plus fine que dans les approches actuelles, nous souhaitons démontrer que les services de transactions construits avec notre canevas logiciel sont bien plus adaptables que les services de transactions actuels sans pour autant impacter les performances à l'exécution (comme nous le démontrons dans la partie III).

La définition d'un canevas logiciel utilisant des composants de très fine granularité nécessite de définir un outillage adapté afin de faciliter la tâche du développeur. Dans la mesure où les composants présentent une granularité plus fine que les approches traditionnelles, ceux-ci sont également présents en plus grand nombre. Cette décomposition favorise non seulement la réutilisation de fonctionnalités de par leur atomicité et leur généricité mais elle permet également de privilégier la description des architectures à leur programmation.

Cependant, la multiplication des composants et la réduction de leur granularité rend la tâche du développeur de canevas logiciel bien plus difficile car celui-ci doit maîtriser l'évolution des nombreux artefacts (interfaces de programmation, implantation des composants, descripteurs de composants, descripteurs d'assemblage, etc.) associés au canevas logiciel. Cette tâche est généralement très coûteuse et en particulier le maintien de la cohérence entre les différents artefacts est souvent sujet à de nombreuses erreurs.

La maîtrise du développement d'un tel canevas logiciel requiert la définition d'une démarche de construction adaptée. En nous inspirant de la philosophie des *exo-noyaux* [EKJ95], nous proposons d'enrichir l'approche qui a été définie dans le canevas logiciel DREAM [Qué05] avec de nouveaux éléments. Le canevas logiciel DREAM propose un canevas logiciel à composants constitué de trois parties :

- un *modèle de composants* généraliste permettant de construire des architectures flexibles et administrables. Plus précisément, nous utilisons le modèle de composants FRACTAL présenté dans la section 3.3.1 du chapitre 3.
- une *bibliothèque de composants* construits avec le modèle de composants généraliste et dédiés à un domaine applicatif particulier. Ces composants peuvent être ensuite assemblés pour former des personnalités.
- des *outils de gestion de configuration* permettant de maîtriser les différentes étapes du cycle de vie des personnalités construites à partir de la bibliothèque de composants. Ces étapes prennent en compte la description, la configuration, le déploiement et l'administration des personnalités.

Notre canevas logiciel GOTM conserve ces trois parties et les complète en définissant :

- un *modèle de programmation* concis pour simplifier et fiabiliser l'utilisation du modèle de composants généraliste, appelé FRACLET. Ce modèle de programmation se détache des abstractions fonctionnelles imposées par le modèle de composants généraliste et propose d'automatiser son support et d'assurer le maintien de la cohérence entre certains artefacts.
- un *langage de description et de vérification de motifs d'architecture* pour capitaliser les constructions récurrentes présentes dans les architectures et automatiser leurs définitions. En appliquant la notion d'invariant d'architecture, nous contrôlons et fiabilisons le processus de génération automatique lors de la description des architectures.
- une *bibliothèque de composants structurée sous forme de motifs de conception* dédiée au domaine applicatif du transactionnel. Les différentes personnalités produites à partir de cette bibliothèque correspondent à des instances de services de transactions répondant à un standard transactionnel déterminé, un modèle de transactions, un protocole de validation ou une combinaison de ces derniers.
- des *modèles de haut-niveau de configuration d'architecture* dédiés au domaine applicatif considéré. Ces modèles utilisent des formalismes plus intelligibles que les descripteurs d'architecture pour les utilisateurs du canevas logiciel.

4.3 Vers des composants de très fine granularité

Afin de bénéficier au mieux de la composition dans les architectures à composants, nous avons opté pour une approche de développement de composants de très fine granularité. Nous présentons dans cette section les principes de conception de ces composants de très fine granularité et nous illustrons cette approche sur l'ingénierie d'une fonction de journalisation.

4.3.1 Principes de conception des composants de très fine granularité

Les composants de très fine granularité correspondent à un sous-ensemble de la conception par composants traditionnelle en appliquant un certain nombre de contraintes liées au développement des infrastructures logicielles radicalement configurables [Qué05]. Notre approche se propose donc de formaliser la conception d'un intergiciel en définissant les règles qui favoriseront son extensibilité et sa configuration.

Poids du composant. Le développement d'un composant de très fine granularité s'appuie sur la définition de micro-interfaces. Nous définissons une micro-interface comme une interface fonctionnelle définissant un faible nombre de méthodes. La définition des micro-interfaces est dirigée par l'identification des groupes de fonctions fournies usuellement par un composant. Ces groupes de fonctions sont réifiés sous la forme de micro-interfaces et sont implantés par des micro-composants. Le poids d'un composant est par conséquent proportionnel au nombre de micro-interfaces qu'il fournit.

Opacité du composant. L'opacité du composant de très fine granularité correspond à son degré de configuration. Ce degré dépend du nombre d'attributs configurables disponibles sur ce composant. En effet, les attributs permettent d'influer sur le comportement du composant. Cette caractérisation est à rapprocher des notions de composants *Blackboxes*, *Whiteboxes* et *Grayboxes* citées dans la littérature [SGM02]. L'opacité évalue par conséquent le degré de contrôle disponible sur un composant de très fine granularité.

Généricité du composant. La généralité du composant de très fine granularité vise à maximiser sa réutilisation. Cette généricité est une conséquence directe du poids et de l'opacité d'un composant. En augmentant la généricité d'un composant, il est possible de le réutiliser dans différents contextes en le configurant différemment. Cette approche favorise la composition des composants de très fine granularité. Dans le contexte du développement d'un canevas logiciel à base de

composants, la composition des composants tient une place prépondérante et contribuant à sa qualité.

Par conséquent, l'objectif idéal de notre démarche est de concevoir des composants légers, transparents et génériques afin de maximiser les capacités d'adaptation de notre canevas transactionnel. Le rôle du langage de description d'architecture devient ainsi prépondérant car la réalisation d'une fonctionnalité ne dépend plus uniquement d'un composant monolithique mais de la composition d'une multitude de composants de très fine granularité. Cette composition détermine la sémantique de la fonctionnalité et offre ainsi la possibilité d'adapter finement cette sémantique en fonction du contexte d'exécution.

4.3.2 Illustration de notre démarche de construction

Nous illustrons les principes de notre démarche de construction à granularité extrêmement fine sur l'ingénierie d'une fonction de journalisation. La journalisation consiste à mémoriser une succession de faits dans des logs (supports de stockages stables). La figure 4.2 fournit une représentation UML de l'implémentation d'une fonction de journalisation en utilisant la programmation objet. L'objet `LoggingImpl` qui implémente l'interface `Logging` est généralement utilisé par les services de transactions pour assurer la reprise sur panne du service en cas de panne inopinée du système. L'opération `write` mémorise dans les logs la valeur du paramètre `data`. Il existe deux types d'opérations d'écriture : *force* ou *non-force*. Les écritures forcées sont mémorisées immédiatement dans les logs tandis que les écritures non-forcées sont stockées temporairement dans un tampon. Le type d'écriture dépend du paramètre `force`. L'opération `read` est utilisée lors du processus de reprise sur panne pour analyser la progression des transactions actives lors de la panne système.

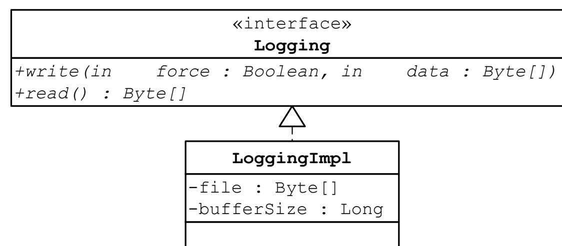


FIG. 4.2: Implantation de la fonction de journalisation en objet.

Si ce type d'implantation de la fonction de journalisation est très répandue [Gir04], celle-ci demeure néanmoins peu extensible d'un point de vue fonctionnel. En observant la sémantique de ces deux opérations, il est possible de structurer différemment cette fonction en utilisant des composants de granularité plus fine que l'objet. La figure 4.3 présente une implantation similaire de la fonction de journalisation en utilisant des composants de granularité fine. Étant donné que les fonctions de lecture et d'écriture dans les logs sont utilisées dans des contextes différents, les deux méthodes sont isolées dans des interfaces différentes (`LoggingReader` et `LoggingWriter`, respectivement). Ensuite, la sémantique de l'opération `write` dépend essentiellement de son paramètre `force`. Afin de mieux séparer la sémantique de l'opération de sa signature, ce paramètre est supprimé et remplacé par deux implantations différentes de l'interface `LoggingWriter` appelées `ForceLoggingWriter` et `NonForceLoggingWriter`. Le composant `NonForceLoggingWriter` dispose d'un attribut `bufferSize` permettant de réaliser des fonctions de log non-forcées avec des tampons de taille variable. Enfin, le code source commun aux fonctions d'écriture et de lecture est placé dans le composant `LoggingProviderImpl` qui implémente les interfaces `LoggingProvider` et `LoggingReader`.

La figure 4.4 propose une composition de la fonction de journalisation exploitant les capacités de partage et de hiérarchisation offertes par le modèle de compo-

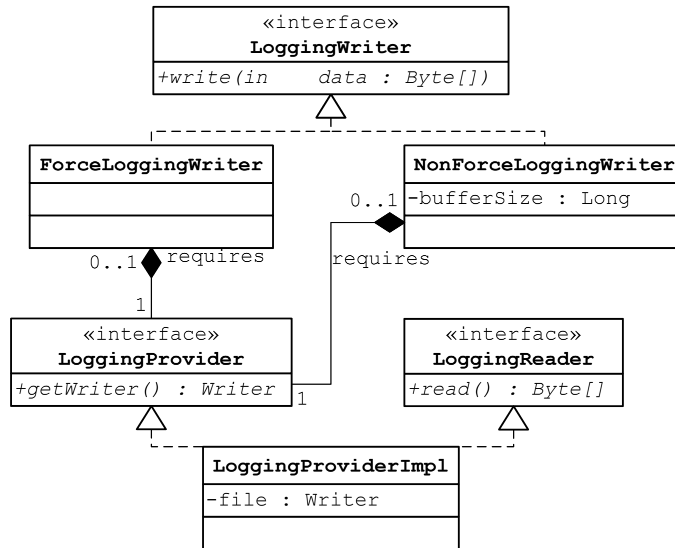


FIG. 4.3: Implantation de la fonction de journalisation en composants de fine granularité.

sants FRACTAL. Le composant `LoggingProviderImpl` est partagé entre les composants `NonForceLogging` et `ForceLogging` qui intègrent les composants `NonForceLoggingImpl` et `ForceLoggingImpl`, respectivement. Ce type de structure facilite l'introduction de nouvelles fonctionnalités dans le composant de journalisation. Par exemple, il est possible d'ajouter un composant de type `NonForceLogging` avec un tampon d'une capacité différente de celui présent dans l'architecture par défaut.

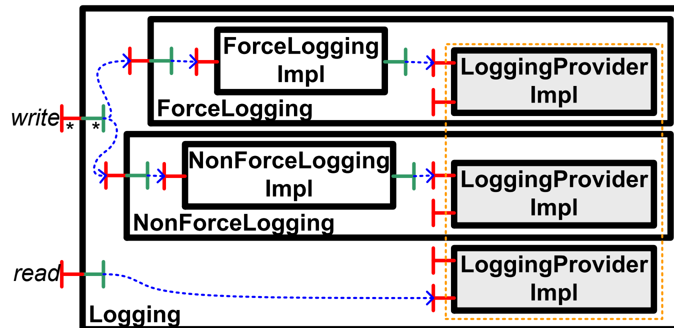


FIG. 4.4: Représentation de la fonction de journalisation avec FRACTAL.

Cette architecture peut être décrite en utilisant le langage de description d'architecture FRACTAL ADL. Cette description d'architecture assemble les composants de journalisation pour réaliser la fonction de journalisation finale nommée `Logging` (ligne 1). Dans un premier temps, les interfaces `read` et `write` du composant sont définies (lignes 2–3). Ensuite, l'architecture du composant `provider` est décrite (lignes 5–8). Les composants `force` (lignes 10–18) et `non-force` (lignes 20–31) représentent des composites partageant leur instance de composant `provider` avec celle du composant `provider` disponible dans le composant `Logging`. Enfin, les interfaces des composants `provider`, `force` et `non-force` sont exportées via les interfaces définies pour le composant `Logging` (lignes 33–35).

L'application de cette décomposition fonctionnelle d'une fonctionnalité objet en un ensemble de composants de plus fine granularité soulève différents problèmes. Tout d'abord, il est nécessaire de développer un plus grand nombre d'artefacts dans notre approche qu'en appliquant l'approche objet. En effet, nous définissons trois composants primitifs et trois interfaces

```
1 <definition name="Logging" arguments="buffer">
2   <interface name="read"  signature="LoggingReader" role ="server" cardinality="singleton"/>
3   <interface name="write" signature="LoggingWriter" role ="server" cardinality="collection"/>

5   <component name="provider">
6     <interface name="read"    signature="LoggingReader" role ="server" cardinality="singleton"/>
7     <interface name="provide" signature="LoggingProvider" role ="server" cardinality="singleton"/>
8     <content class="LoggingProviderImpl"/>
9   </component>

11  <component name="force">
12    <interface name="write"    signature="LoggingWriter" role ="server" cardinality="singleton"/>
13    <component name="writer">
14      <interface name="write"   signature="LoggingWriter" role ="server" cardinality="singleton"/>
15      <interface name="provide" signature="LoggingProvider" role ="client" cardinality="singleton"/>
16      <content class="ForceLoggingImpl"/>
17    </component>
18    <component name="provider" definition="./provider"/>
19    <binding client="this.write"     server="writer.write"/>
20    <binding client="writer.provide" server="provider.provide"/>
21  </component>

23  <component name="non-force">
24    <interface name="write" signature="LoggingWriter" role ="server" cardinality="singleton"/>
25    <component name="writer">
26      <interface name="write"   signature="LoggingWriter" role ="server" cardinality="singleton"/>
27      <interface name="provide" signature="LoggingProvider" role ="client" cardinality="singleton"/>
28      <content class="NonForceLoggingImpl"/>
29      <attributes signature="NonForceLoggingAttributes">
30        <attribute name="buffer" value="${buffer}"/>
31      </attributes>
32    </component>
33    <component name="provider" definition="./provider"/>
34    <binding client="this.write"     server="writer.write"/>
35    <binding client="writer.provide" server="provider.provide"/>
36  </component>

38  <binding client="this.read"            server="provider.read"/>
39  <binding client="this.write-force"     server="force.write"/>
40  <binding client="this.write-non-force" server="non-force.write"/>
41 </definition>
```

LST. 4.1: Description FRACTAL ADL du composant Logging.

pour modéliser la fonctionnalité de journalisation alors qu'il suffit d'une seule interface et d'un seul objet pour réaliser cette même fonction en objet. Ensuite, le développement de la fonctionnalité avec un modèle de composants est *a priori* plus coûteux qu'un développement en objet car il est nécessaire de supporter les propriétés techniques imposées par le modèle de composants, c.-à-d. implanter les interfaces de contrôle. Il est également nécessaire de décrire exhaustivement tous les composants développés dans des descripteurs d'architecture en utilisant un langage de description d'architecture donné. Enfin, il faut décrire les assemblages de ces composants nécessaires à la réalisation de la fonctionnalité comme illustré dans le listing 4.1. Ces assemblages peuvent dupliquer certaines constructions telles que le partage des instances de composants (lignes 18 et 33) ou l'exportation des interfaces serveur d'un composant (lignes 19 et 34). De la même manière, des motifs d'assemblage, comme la liaison des interfaces multiples, présentent de fortes similarités (lignes 39–40).

Pour remédier à ces problèmes, nous proposons d'utiliser notre modèle de programmation et notre langage de description de motifs d'architecture pour simplifier et fiabiliser le développement des intergiciels à base de composants.

4.4 Présentation du canevas logiciel GOTM

Cette section présente la démarche de construction que nous avons développée dans le cadre du canevas logiciel GOTM. Cette démarche de contruction est composée de quatre contributions. La première contribution consiste en un modèle de programmation qui simplifie et fiabilise le développement des composants. La seconde contribution est un langage de description et de vérification de motifs d'architecture. Nous présentons ensuite la bibliothèque de composants dédiés au domaine du transactionnel construite à l'aide de notre modèle de programmation et de notre langage de description. Enfin, cette bibliothèque peut être configurée à l'aide de différents modèles de haut niveau que nous avons définis dans notre quatrième contribution.

4.4.1 Démarche de construction de GOTM

La figure 4.5 illustre la démarche de construction des services intergiciels que nous proposons. Cette démarche identifie trois rôles : le **développeur**, l'**architecte** et l'**intégrateur**. Le développeur utilise notre modèle de programmation pour développer les composants de la bibliothèque. L'architecte s'appuie sur le langage de description et de vérification de motifs d'architecture pour définir des motifs utilisant les composants de la bibliothèque. Enfin, l'intégrateur configure les motifs d'architectures disponibles en utilisant différents modèles de haut niveau. Les transitions entre les rôles sont assurées par des outils de génération et de transformation des artefacts afin d'aboutir à la réalisation d'un service intergiciel adapté aux besoins de l'utilisateur final.

Les trois rôles identifiés par cette démarche correspondent également à trois niveaux de modélisation. Les *modèles de configuration* associés au rôle d'intégrateur fournissent une vision abstraite et intelligible du comportement des services de transactions. Les *motifs d'architecture* associés au rôle d'architecte fournissent une vision structurelle très fine de l'architecture des services de transactions. Enfin, la *bibliothèque de composants* associée au rôle du développeur encapsule les aspects techniques liés à la réalisation des services de transactions.

4.4.2 Modèle de programmation des composants

Le développement de notre canevas logiciel requiert l'utilisation d'un modèle de composants généraliste fournissant un *modèle de programmation fiable*, nommé FRACLET, et un *modèle de composition étendu*. Le modèle de composants FRACTAL supporte la création de structures hiérarchiques avec partage de composants. De plus, celui-ci dispose d'un modèle de programmation modulaire basé sur le langage de programmation Java.

Nous proposons donc d'étendre le modèle de programmation des composants FRACTAL pour simplifier et fiabiliser le développement des composants de notre canevas logiciel. Notre modèle

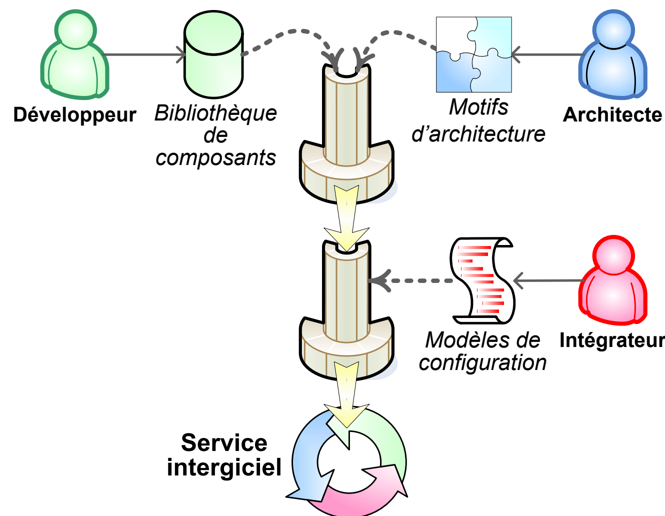


FIG. 4.5: Présentation de la démarche GOTM de construction d'intergiciels.

de programmation FRACLET adresse les problèmes de *séparation des préoccupations techniques et fonctionnelles* et de *redondance de certaines méta-informations* qui pénalisent les développements à base de composants. Afin de résoudre ces problèmes, nous proposons d'appliquer les principes de la programmation par attributs au développement de composants basés sur le modèle de composants FRACTAL. Ainsi, nous réifions les éléments caractéristiques du modèle de programmation sous la forme d'un ensemble d'annotations dédiées à la programmation par composants. Ces annotations sont appliquées sur le code fonctionnel des composants et remplacent le code technique imposé par le modèle de composants FRACTAL. Des générateurs interprêtent ensuite le code annoté et génèrent automatiquement non seulement le code technique des composants mais aussi leurs descriptions architecturales.

L'utilisation du modèle de programmation FRACLET permet de réduire drastiquement la taille du code source de l'application en isolant le code fonctionnel. De plus, l'utilisation des générateurs assure le maintien de la cohérence entre le code source de l'application et les différents descripteurs représentant la méta-information des composants.

4.4.3 Langage de description et de vérification de motifs d'architecture

La description à granularité fine de l'architecture d'un service de transactions nécessite un *langage de description d'architecture modulaire et fiable* afin de valider les assemblages décrits par l'architecte. Le modèle de composants FRACTAL dispose d'un langage de description d'architecture appelé FRACTAL ADL. Ce langage supporte la description modulaire d'une architecture complexe à base de composants.

Nous proposons donc d'étendre le langage FRACTAL ADL et son usine pour y intégrer deux nouveaux opérateurs de description et de vérification de motifs d'architecture. Ces deux opérateurs visent à factoriser les constructions récurrentes présentes dans la définition des architectures et à améliorer la séparation des préoccupations prises en compte dans la définition d'une architecture. Ces opérateurs exploitent la syntaxe FPATH [Dav05] afin de pouvoir naviguer facilement dans l'architecture et ainsi capturer les éléments qui leur semblent pertinents. Le premier opérateur que nous définissons permet de générer à la volée des constructions de l'architecture (liaisons, définitions de composants, etc.). Le second opérateur nous permet de définir des invariants d'architecture qui garantissent la bonne application des règles architecturales définies par l'architecte.

L'utilisation de ce langage de description et de vérification de motifs d'architecture permet de réduire la taille des descriptions de l'architecture. Les descriptions obtenues sont suffisamment

modulaires pour être facilement réutilisées dans d'autres contextes.

4.4.4 Bibliothèque de composants transactionnels GOTM

La bibliothèque de composants transactionnels capitalise les grandes fonctionnalités requises lors de la construction d'un service de transactions. En nous appuyant sur notre langage de description et de vérification de motifs d'architecture, nous identifions une *architecture abstraite d'un service de transactions* composée d'une partie statique et d'une partie dynamique. Les éléments caractéristiques de cette architecture correspondent à des *motifs de conception* réifiés sous forme de composants. Ces composants sont hautement configurables car les motifs de conception qu'ils représentent sont définis pour être facilement étendus et adaptés aux besoins du service.

La bibliothèque de composants transactionnels fournit également un ensemble de composants encapsulant les fonctions communes présentes dans de nombreux services de transactions. Ces composants sont implantés en utilisant notre modèle de programmation de composants FRACTAL. Par exemple, GOTM fournit un *gestionnaire d'états* pour assurer la consistance de l'exécution d'une transaction, un *bus de messages* pour propager les états de la transaction, des *protocoles de validation* pour assurer l'atomicité des transactions et des *gestionnaires de participants* pour supporter différents types de ressources transactionnelles. Chacun de ces composants peut être configuré de différentes manières pour réaliser une personnalité déterminée d'un service de transactions.

L'utilisation d'un canevas logiciel à base de composants de très fine granularité pour construire des services de transactions offre une plus grande adaptabilité structurelle et comportementale des services sans nuire à leurs performances.

4.4.5 Modèles dédiés à la configuration des services de transactions

Enfin, la configuration d'un canevas logiciel à base de composants demeure une tâche complexe que seuls les architectes du canevas sont en mesure de maîtriser. En effet, la construction et la configuration d'une personnalité à partir du canevas logiciel se repose sur le langage de description d'architecture utilisé par ce dernier. Or, l'intégrateur de la personnalité ne dispose pas toujours des connaissances suffisantes pour configurer finement et efficacement les caractéristiques d'une personnalité.

Nous proposons donc d'utiliser des modèles de haut niveau pour faciliter la construction et la configuration des personnalités basées sur notre canevas logiciel. Ces modèles de haut niveau font abstraction de la technologie sous-jacente utilisée pour réaliser le service de transactions. Par exemple, dans le cadre de notre canevas logiciel GOTM, nous proposons d'utiliser les *diagrammes UML de séquences et d'états* pour décrire le modèle de transactions. Nous utilisons également des règles de type *Événement–Condition–Action* pour décrire le comportement des participants d'une transaction. Enfin, nous définissons un méta-modèle dédié à la configuration du service de transaction et de ses différentes caractéristiques. Chacun de ces modèles identifie une caractéristique particulière du service de transactions. Les configurations exprimées avec ces modèles sont ensuite transformées en différents assemblages des composants du canevas logiciel GOTM.

L'utilisation de ces modèles de haut niveau fournit une représentation intelligible pour l'intégrateur des caractéristiques du service de transactions. Ces modèles capitalisent les caractéristiques des services de transactions indépendamment des technologies de mise en œuvre.

4.5 Organisation de la deuxième partie

La seconde partie du document s'organise de la façon suivante. Le chapitre 5 introduit notre modèle de programmation des composants FRACTAL nommé FRACLET. Le chapitre 6 présente notre langage de description et de vérification de motifs d'architecture. Le chapitre 7 détaille la bibliothèque de composants transactionnels nommée GOTM. Enfin, le chapitre 8 propose les quatre modèles de configuration que nous avons définis pour le canevas logiciel GOTM.

Every non-trivial program can be simplified by at least one line of code.

Murphy's Law

Chapitre 5

Le modèle de programmation FRACLET pour les composants GOTM

Sommaire

5.1 Motivations **74**
- 5.1.1 Entrelacement du code métier et technique 74
- 5.1.2 Redondance des méta-informations 76

5.2 Principes de la programmation par attributs **77**

5.3 Application au modèle de composants FRACTAL **78**
- 5.3.1 Description des annotations FRACLET 78
- 5.3.2 Description des générateurs FRACLET 80
- 5.3.3 Application HelloWorld revisitée 81

5.4 Éléments d'implantation de FRACLET **83**
- 5.4.1 Description du moteur de génération 83
- 5.4.2 Illustration du code généré 84

5.5 Évaluations **87**
- 5.5.1 Évaluation quantitative du code source 87
- 5.5.2 Évaluation quantitative du code généré 88
- 5.5.3 Évaluation qualitative 89

5.6 Généralisation à d'autres modèles de composants **90**
- 5.6.1 Programmation orientée composant et dirigée par le code 90
- 5.6.2 Description des générateurs OPENCOM 91
- 5.6.3 Illustration du code généré pour OPENCOM 91
- 5.6.4 Évaluation 91

5.7 Travaux connexes **94**
- 5.7.1 Programmation par attributs 94
- 5.7.2 Programmation générative 94
- 5.7.3 Programmation par aspects 95
- 5.7.4 Langages dédiés 95

5.8 Conclusion **96**

Ce chapitre présente Fraclet, *le modèle de programmation utilisé pour développer les composants* GoTM*. L'objectif de* Fraclet *est de simplifier le développement des applications à base de composants. Pour ce faire, nous utilisons la programmation par attributs pour marquer le code métier du composant avec les concepts relatifs à la programmation par composants. Les artefacts requis par l'utilisation d'un modèle de composants concret sont produits* a posteriori *par des générateurs.*
Après avoir identifié les motivations de cette approche et avoir introduit les principes de la programmation par attributs, nous présentons Fraclet *—le modèle de programmation des composants* Fractal *et son implantation.* Fraclet *est ensuite généralisé pour être appliqué également sur le modèle de composants* OpenCOM*. Cette généralisation permet d'exécuter les composants* GoTM *sur les substrats d'exécution disponibles pour les modèles de composants* Fractal *et* OpenCOM*.*

5.1 Motivations

Le paradigme de programmation par composants est désormais reconnu dans le domaine de l'ingénierie logicielle pour sa capacité à améliorer la productivité, la réutilisation et la composition [SGM02]. Bien que bénéficiant d'un intérêt grandissant, la programmation par composants demeure difficile à appréhender par les développeurs. En effet, la programmation d'un composant nécessite de compléter le code métier avec du code technique spécifique au modèle de composants utilisé. Ce code technique est non seulement répétitif et verbeux mais il est souvent sujet à de nombreuses erreurs. Par exemple, lors du développement d'un composant avec le modèle Fractal, le code technique consiste en l'implantation de différentes interfaces de contrôle (liaison, attribut, cycle de vie, etc.) [BCL+06]. De plus, un composant logiciel est généralement accompagné de sa description dans un langage de description d'architecture (ADL pour *Architecture Description Language*) particulier afin de faciliter son intégration dans un système donné. Cette description reprend en partie les informations décrites dans le composant logiciel, introduisant ainsi de la redondance entre la description et le code d'un composant logiciel donné.

5.1.1 Entrelacement du code métier et technique

Bien que la programmation par composants améliore la modularité, la configuration et la réutilisation des applications, l'utilisation d'un modèle de composants particulier implique également plus de complexité et plus de redondance dans l'information exprimée par le développeur. Cependant, cette complexité dépend principalement du modèle de programmation utilisé pour développer l'application. Ce modèle de programmation met en œuvre un modèle abstrait dans un langage de programmation donné. Par conséquent, les concepts du modèle abstrait sont enfouis dans le code métier. De plus, en introduisant des dépendances vers le modèle de composants, le code mélange le métier et les propriétés techniques. Pour illustrer ce problème, le développement de l'application HelloWorld présenté dans la figure 5.1 est détaillé dans cette section. Le composant HelloWorld se compose de deux composants primitifs (composants Client et Server) connectés entre eux et d'un composant composite (composant HelloWorld) qui exporte l'interface serveur du composant client. Le composant serveur dispose d'un attribut header afin de configurer l'entête des messages qu'il affiche.

L'interface `Service` décrite dans le listing 5.1 comporte la seule méthode `print` spécifique au métier de l'application (ligne 2).

Le composant `Client`, décrit dans le listing 5.2, initialise le message de bienvenue lors du démarrage du composant `Client` en utilisant l'interface `LifeCycleController` du modèle de composants (lignes 3–6). Ensuite, l'interface `Service` requise est résolue en utilisant l'interface `BindingController` définie par Fractal (lignes 8-22). Finalement, le code métier du

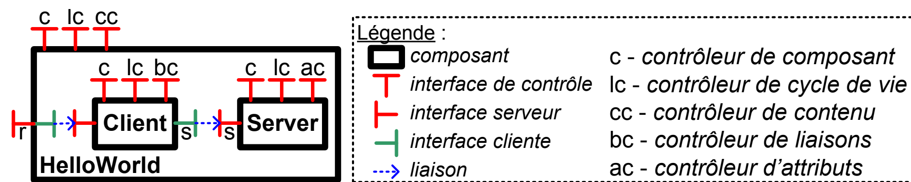


FIG. 5.1: Architecture FRACTAL du composant HelloWorld.

```
1 public interface Service {
2   void print(String message);
3 }
```

LST. 5.1: Description de l'interface Service.

composant `Client` est décrit par la méthode `run` (ligne 25).

```
1 public class Client implements Runnable, LifeCycleController, BindingController {
2   // Interface LifeCycleController
3   protected String message;
4   public String getFcState() { return null; }
5   public void startFc() { this.message = "Hello world!"; }
6   public void stopFc() { }

8   // Interface BindingController
9   protected Service s; // interface Service requise
10  public String[] listFc() { return new String[] { "s" }; }
11  public Object lookupFc(String name) throws NoSuchInterfaceException {
12    if (name.equals("s")) return this.s;
13    throw new NoSuchInterfaceException(name+" is unknown!");
14  }
15  public void bindFc(String name, Object ref) throws NoSuchInterfaceException {
16    if (name.equals("s")) this.s = (Service) ref;
17    else throw new NoSuchInterfaceException(name+" is unknown!");
18  }
19  public void unbindFc(String name) throws NoSuchInterfaceException {
20    if (name.equals("s")) this.s = null;
21    else throw new NoSuchInterfaceException(name+" is unknown!");
22  }

24  // Interface Runnable
25  public void run() { this.s.print(this.message); }
26 }
```

LST. 5.2: Mise en œuvre du composant FRACTAL Client.

Le composant Server peut être configuré via un attribut header. Pour cela, une interface de contrôle `ServerAttributes` doit donc être définie (listing 5.3). Cette interface définit deux méthodes `getHeader()` et `setHeader()` pour configurer la valeur de l'attribut du composant.

Le composant `Server` doit décrire dans un premier temps la gestion de l'attribut `header` en implantant l'interface `ServerAttributes` (lignes 3–5 du listing 5.4). La configuration du mécanisme de trace est ensuite assurée par l'implantation de l'interface `Loggable` (lignes 7–12). Finalement, le code métier du composant `Server` est décrit dans la méthode `print` (lignes 15–17).

Discussion. Le constat qui apparaît à la lecture du code décrit dans les listings 5.1 à 5.4 est que le développement de composants FRACTAL est très verbeux. En effet, sur les 50 lignes de code de l'application, 30 lignes sont liées au code technique alors que seulement 20 lignes adressent le code métier (déclaration des champs et méthodes métiers). Ceci signifie que 66% du code

```
1 public interface ServerAttributes extends AttributeController {
2   String getHeader();
3   void setHeader(String value);
4 }
```

LST. 5.3: Description de l'interface de contrôle FRACTAL ServerAttributes.

```
1 public class Server implements Service, ServerAttributes, Loggable {
2   // Interface ServerAttributes
3   protected String header; // attribut header.
4   public String getHeader () { return this.header; }
5   public void setHeader (String value) { this.header = value; }

7   // Interface Loggable
8   protected Logger logger; // service de trace.
9   public void setLogger(Logger value) { this.logger = value; }
10  public Logger getLogger() { return this.logger; }
11  public void setLoggerFactory(LoggerFactory value) { }
12  public LoggerFactory getLoggerFactory() { return null; }

14  // Interface Service
15  public void print(String msg) {
16    this.logger.info(this.header + msg);
17 } }
```

LST. 5.4: Mise en œuvre du composant FRACTAL Server.

de l'application est imposé par l'utilisation du modèle de composants[1]. Ce code n'est pas intuitif et facilement sujet à erreurs. En outre, toute modification de la structure du composant doit impacter le code technique de façon cohérente. Par exemple, la définition d'un nouvel attribut requiert la modification de l'interface de contrôle et du composant pour y intégrer le code technique associé à ce nouvel attribut. L'ajout ou la modification d'une interface cliente d'un composant impacte également plusieurs méthodes du code technique associé au composant. Finalement, toute évolution du modèle de composants, telle que des changements dans les interfaces de contrôle, implique une modification de tous les composants précédemment développés. L'objectif de FRACLET au regard de cette limitation est de réduire la taille du code écrit par le développeur. Pour cela, le code technique des composants FRACTAL est remplacé par un ensemble d'annotations capturant la sémantique associée aux spécificités de FRACTAL. Des générateurs se chargent ensuite de produire du code technique cohérent avec la structure du composant. Dès lors, l'évolution du modèle de composants impacte uniquement les générateurs de FRACLET et non plus le code de toutes les applications FRACTAL.

5.1.2 Redondance des méta-informations

FRACTAL ADL facilite la description et la composition des composants d'une application. Le composant Client est décrit dans le listing 5.5 en utilisant cet ADL. Cette description précise l'interface `Runnable` fournie par le composant (ligne 2), l'interface `Service` requise (ligne 3) et le contenu du composant, c.-à-d. la classe d'implantation du composant (ligne 4).

```
1 <definition name="Client">
2   <interface name="r" role="server" signature="java.lang.Runnable"/>
3   <interface name="s" role="client" signature="Service"/>
4   <content class="Client"/>
5 </definition>
```

LST. 5.5: Description FRACTAL ADL du composant Client.

[1] Ce pourcentage dépend fortement de l'application considérée.

De façon similaire, le composant Server précise, dans le listing 5.6, l'interface `Service` qu'il fournit (ligne 2) et son contenu (ligne 3). À cela s'ajoute la description de l'attribut header du composant dont la valeur sera fixée lors de l'instanciation via l'utilisation d'un argument (lignes 4–6).

```
1 <definition name="Server" arguments="header">
2   <interface name="s" role="server" signature="Service"/>
3   <content class="Server"/>
4   <attributes signature="ServerAttributes">
5     <attribute name="header" value="${header}"/>
6   </attributes>
7 </definition>
```

LST. 5.6: Description FRACTAL ADL du composant Server.

Enfin, le composant HelloWorld est décrit dans le listing 5.7. Ce composant contient un composant `Client` et un composant `Server` liés par l'interface `Service` (lignes 3–5). L'interface `Runnable` fournie par le composite (ligne 2) est liée à l'interface `Runnable` fournie par le composant `Client` (ligne 6).

```
1 <definition name="HelloWorld">
2   <interface name="r" role="server" signature="java.lang.Runnable"/>
3   <component name="client" definition="Client"/>
4   <component name="server" definition="Server(->)"/>
5   <binding client="client.s" server="server.s"/>
6   <binding client="this.r" server="client.r"/>
7 </definition>
```

LST. 5.7: Description FRACTAL ADL du composant HelloWorld.

Discussion. Le constat à l'issue de l'observation des fichiers FRACTAL ADL est que la plupart des informations sont redondantes avec celles contenues dans le code de l'application précédemment détaillée. En effet, les informations exprimées dans les listings 5.5 et 5.6 ne font que compléter le code de l'application en précisant le nom des interfaces fournies et les arguments du composant. Cette description non seulement verbeuse peut également être sujette à de nombreuses incohérences lors de la modification de la structure du composant. Seul le composant HelloWorld décrit dans le listing 5.7 comporte des informations absentes du code de l'application. En effet, il précise les composants mis en jeu dans l'application ainsi que leurs liaisons et la valeur de leurs arguments. L'objectif de FRACLET au regard de cette limitation est de produire automatiquement les fichiers FRACTAL ADL associés aux composants primitifs. Pour cela, le code de ces composants doit être annoté avec les méta-informations présentes dans les fichiers FRACTAL ADL mais non exprimées dans le code de l'application. Les descriptions des composants primitifs générées automatiquement par FRACLET complètent ainsi la configuration de l'application globale.

5.2 Principes de la programmation par attributs

La programmation par attributs (@OP pour *Attribute-Oriented Programming*) est une technique de marquage de code [Paw05, WR03]. Plus précisément, le développeur peut marquer les éléments du programme (par exemple, classes, méthodes et champs) avec des *annotations* pour attacher à ces éléments des sémantiques spécifiques à l'application ou au domaine [ESM05, WS05]. Par exemple, le développeur peut définir une annotation de trace et l'associer à une méthode pour indiquer que les appels à cette méthode devront être tracés. Un développeur différent peut définir une annotation service Web et l'associer à une classe pour préciser que cette classe implante un service web. Les annotations permettent de séparer la logique applicative de la logique

liée à l'intergiciel ou au domaine. En cachant les détails sémantiques de mise en œuvre du code métier, les annotations augmentent le niveau d'abstraction de programmation tout en réduisant la complexité de programmation. Par conséquent, le code devient plus simple et plus lisible.

La notion d'annotation peut être rapprochée de la notion de stéréotype dans UML. En effet, un stéréotype permet de marquer des éléments de modèles avec une sémantique particulière. D'ailleurs, certains travaux proposent de transformer automatiquement des éléments UML marqués par des stéréotypes vers un programme source annoté [WSTD05a, WSTD05b].

Les éléments de programme associés à des annotations sont transformés en des programmes plus détaillés par des *générateurs*. Par exemple, un générateur peut introduire du code de trace dans le code d'une méthode marquée par l'annotation de trace.

L'@OP autorise l'*intégration en continu* du code métier pour les développements à base de composants. Le principe d'intégration en continu permet au développeur de générer les artefacts requis par l'intergiciel à n'importe quel moment du développement du composant. Les développeurs se concentrent uniquement sur l'édition d'un seul fichier par composant. Les métadonnées de déploiement sont intégrées en continu sans se soucier de leur mise à jour. Alors que le développement d'un composant se décompose généralement en plusieurs fichiers (7 pour un composant EJB), l'@OP permet au développeur de travailler avec un seul fichier et de générer automatiquement les autres fichiers. Par conséquent, le développeur peut se concentrer sur la logique applicative tout en réduisant drastiquement les temps de développement.

5.3 Application au modèle de composants FRACTAL

Cette section présente l'application de la programmation par attributs sur le modèle de composants FRACTAL. Notre proposition, appelée FRACLET, se compose d'un jeu d'annotations pour la programmation par composants introduit dans la section 5.3.1 et d'un ensemble de générateurs pour le modèle de composants FRACTAL présenté dans la section 5.3.2. Afin d'illustrer les bénéfices de notre approche, le code de l'application HelloWorld décrit dans la section 5.1.1 est revisité dans la section 5.3.3 en utilisant FRACLET.

5.3.1 Description des annotations FRACLET

Les annotations sont intégrées au code de l'application pour enrichir le langage de programmation avec les concepts du modèle de composants FRACTAL. Il est donc nécessaire d'identifier ces concepts en étudiant le modèle abstrait de FRACTAL. Le modèle présentant les concepts pertinents pour la programmation d'un composant primitif FRACTAL est décrit dans la figure 5.2. Un composant fournit et requiert un certain nombre d'interfaces. Ces interfaces sont caractérisées par un nom, une signature, une contingence (obligatoire ou optionnelle) et une cardinalité (singleton ou collection). Un composant peut définir des attributs identifiés par un nom et initialisés par une valeur. Un composant peut utiliser un contrôleur fourni par le substrat d'exécution du modèle de composants. Parmi ces services, la gestion du cycle de vie peut notifier le composant de l'évolution de son état, c.-à-d. créé, démarré, stoppé ou détruit.

À partir de cette identification des concepts du modèle de programmation de FRACTAL, il est possible de les réifier en un ensemble d'annotations dédiées à la programmation de composants FRACTAL. Ainsi, le tableau 5.1 résume les huit annotations suffisant à capturer la programmation d'un composant FRACTAL :

1. L'annotation @data marque une classe ou une interface pour préciser que celle-ci représente une structure de données et non pas un composant primitif.
2. L'annotation @component marque une classe pour préciser le nom du composant primitif et le type de membrane qui doit lui être associé.
3. L'annotation @provides marque une classe ou une interface pour préciser la nature des services fournis par celle-ci (nom, signature).

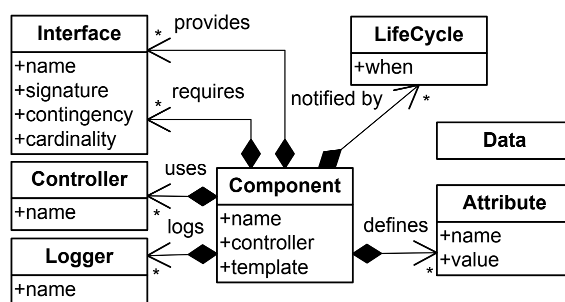


FIG. 5.2: Modèle de programmation FRACLET.

4. L'annotation @requires marque les champs d'une classe pour préciser la nature des services requis par le composant (nom, signature, éventualité, cardinalité).
5. L'annotation @attribute marque les champs d'une classe pour décrire des attributs du composant caractérisés par un nom et une valeur initiale.
6. L'annotation @controller marque les champs d'une classe pour décrire une dépendance vers un contrôleur fourni par la membrane du composant.
7. L'annotation @logger marque les champs d'une classe pour associer un *logger* au composant.
8. L'annotation @lifecycle marque les méthodes d'un composant pour décrire les traitements qui devront être exécutés à certaines étapes du cycle de vie du composant. Les étapes `create` et `destroy` ne sont pas des étapes standards du cycle de vie du modèle de composant FRACTAL mais sont liées à la création et la destruction d'un composant.

Annotation	Élément	Paramètre	Description	Contingence	Valeur par défaut
@data	`classe` ou `interface`			-	-
@component	`classe` ou `interface`	*name* *controller* *template*	nom du composant membrane associée au composant membrane du composant *template*	optionnel optionnel optionnel	signature de la classe primitive primitiveTemplate
@provides	`classe` ou `interface`	*name* *signature*	nom de l'interface serveur signature de l'interface serveur	optionnel optionnel	nom de l'interface signature de l'interface

@requires	`champ`	*name*
		signature
		contingency
		
		cardinality
			nom de l'interface cliente
			signature de l'interface cliente
			éventualité de la liaison
			(`mandatory`|`optional`)
			cardinalité de la liaison
			(`singleton`|`collection`)
				optionnel
				optionnel
				optionnel
				
				optionnel
					nom du champ
					signature du champ
					`mandatory`
					
					`singleton`

@attribute	`champ`	*name* *value* *property*	nom de l'attribut valeur initiale de l'attribut nom de la propriété de configuration	optionnel optionnel optionnel	nom du champ - -
@controller	`champ`	*name*	nom du contrôleur utilisé	optionnel	`component`
@logger	`champ`	*name*	nom du *logger*	optionnel	nom du composant
@lifecycle	`méthode`	*when*	cycle de vie observé (`create`|`start`|`stop`|`destroy`)	requis	-

TAB. 5.1: Jeu d'annotations pour le modèle de programmation de FRACTAL.

Vérifications précoces. L'application des annotations sur le code source d'une application permet d'enrichir et de capturer la sémantique associée aux différents éléments de code des composants. En particulier, il est possible de différencier les attributs Java représentant des références vers des interfaces clientes de ceux représentant un attribut de composant ou encore un état interne. Ainsi, outre la vérification du typage des annotations et de leurs paramètres (décrites dans le tableau 5.1), les annotations FRACLET intègrent également les vérifications précoces suivantes :
– L'annotation @requires ne peut être appliquée que sur un attribut Java de type interface.
– L'annotation @attribute ne peut être appliquée que sur un attribut Java de type primitif.

À terme, ces vérifications précoces doivent également supporter :
- La détection des chemins de communication cachés en s'assurant que les paramètres des méthodes des interfaces de composants sont de type primitif ou annotés par l'annotation @data.
- Le contrôle de la reconfiguration des liaisons en vérifiant que les attributs Java marqués par l'annotation @requires ne sont pas modifiés par d'autres méthodes que les méthodes du contrôleur de liaisons.

Enfin, il est également possible d'intégrer des vérifications liées à l'utilisation d'une technologie ou d'un substrat particulier telles que :
- Dans le modèle de programmation FRACTAL, le composant doit fournir un constructeur par défaut, les interfaces fournies ne doivent pas être figées (en utilisant le mot-clé `final`).
- Dans le modèle de programmation OPENCOM, le composant doit fournir un constructeur dont le paramètre est un objet de type `IUnknown`.

En utilisant ces huits annotations, il est désormais possible de supprimer tout le code technique présent dans les composants afin de réduire non seulement la taille du code mais également les erreurs potentielles liées au développement. Le code technique est introduit *a posteriori* dans le composant par l'utilisation de générateurs. Ces générateurs se chargent également de produire tous les fichiers de description et de configuration nécessaires au bon fonctionnement de l'application.

5.3.2 Description des générateurs FRACLET

Afin de produire les différents artefacts requis par FRACTAL, les générateurs s'appuient sur un moteur de génération pour analyser le code de l'application et générer les différents fichiers. Ce moteur de génération est illustré dans la figure 5.3. Le moteur est paramétré par la définition des annotations à traiter et la liste des générateurs à exécuter. Le code annoté de l'application est placé en entrée du moteur de génération afin de produire les différents artefacts. Les fichiers de code générés (en Java) sont ensuite compilés avec les fichiers originaux alors que les fichiers de description (en XML) sont utilisés par l'outil FRACTAL ADL pour déployer l'application.

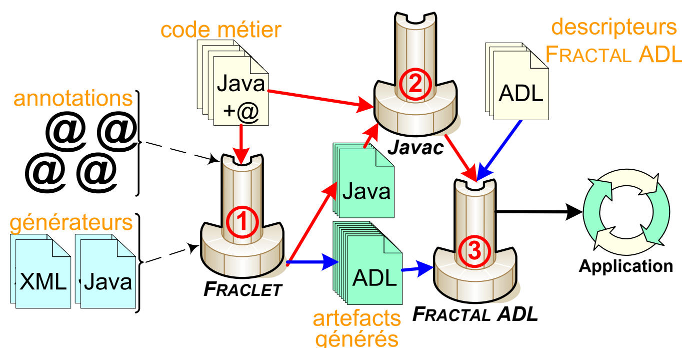


FIG. 5.3: Moteur de génération FRACLET.

FRACLET fournit cinq générateurs pour le modèle de composants FRACTAL :

Contrôleur d'attributs est un générateur de code Java qui, pour tout composant définissant au moins un attribut (avec l'annotation @attribute), génère une interface de contrôle des attributs spécifique au composant. Cette interface hérite de l'interface `AttributeController` imposée par le modèle de composants FRACTAL comme illustré avec l'interface `ServerAttributes` dans la figure 5.4. Si un composant hérite d'un autre composant définissant un ensemble d'attributs, alors son interface de contrôle des attributs hérite de l'interface de contrôle du composant

hérité. Si ce composant n'introduit pas de nouvel attribut, alors il implémente directement l'interface de contrôle du composant hérité. Cette sémantique limite le nombre et la taille des artefacts générés par FRACLET afin d'optimiser l'empreinte mémoire de l'application.

Composant primitif est un générateur de code Java qui, pour tout composant utilisant au moins une des huit annotations définies précédemment, génère le code technique du composant pour le modèle de composants FRACTAL. En cas d'héritage entre composants, le code généré par FRACLET prendra en compte les annotations définies à chaque niveau de l'arborescence d'héritage. Comme illustré avec les classes `FcClient` et `FcServer` dans la figure 5.4, ce code technique inclut la mise en œuvre des interfaces `BindingController` pour la gestion des liaisons, `LifeCycleController` pour la notification du cycle de vie et `Loggable` pour la gestion de la trace. Si le composant définit au moins un attribut, alors le code technique utilisant l'interface produite par le générateur de contrôleur d'attributs est également introduit.

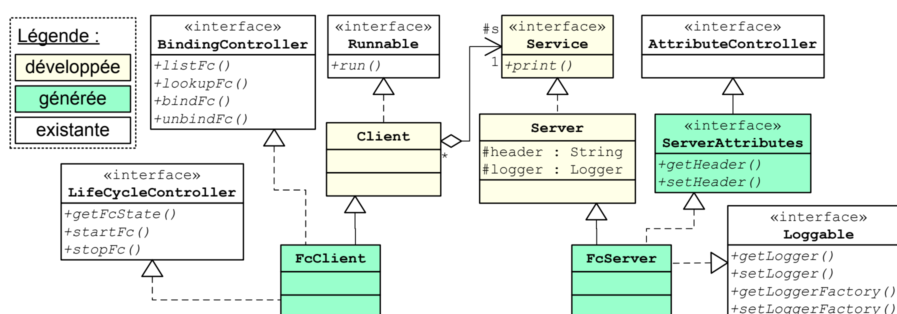


FIG. 5.4: Génération Java pour une application de type client-serveur.

Définition primitive est un générateur de description XML, qui pour tout composant primitif ou toute interface métier, génère une description architecturale de ce composant conforme à la syntaxe FRACTAL ADL. En cas d'héritage entre composants, la description FRACTAL ADL générée par FRACLET étendra la description associée à la classe héritée. Dans la figure 5.5, les descriptions Client et Server correspondent aux descriptions architecturales générées par le générateur de définition primitive pour les composants Client et Server. La description Service est également générée par ce générateur à partir de l'interface métier Service.

Définition composite abstraite est un générateur de description XML, qui pour tout composant primitif définissant au moins une interface requise (avec l'annotation @requires), génère une définition de composant composite incluant la définition du composant primitif considéré par le générateur et la résolution de ses liaisons vers des définitions de composants abstraites. Dans la figure 5.5, la description produite par ce générateur est la description ClientComposite qui associe la description du composant primitif Client à la description abstraite de l'interface Service.

Configuration de trace est un générateur de fichier texte qui génère un fichier de configuration de la trace pour les différents traceurs définis dans le code de l'application. Ce fichier respecte le formalisme des fichiers de configuration défini par le canevas logiciel MONOLOG[2].

5.3.3 Application HelloWorld revisitée

Cette section illustre l'utilisation des huit annotations disponibles pour la programmation de composants FRACTAL. Pour cela, nous revisitons l'application HelloWorld présentée dans la

[2]Le projet Monolog : http://monolog.objectweb.org/

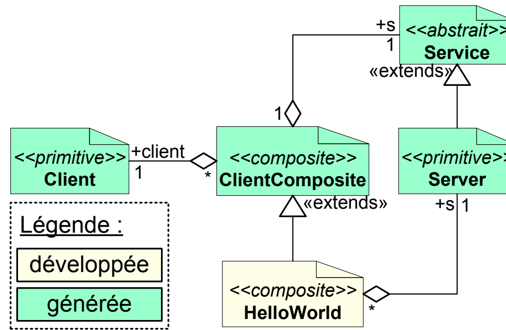


FIG. 5.5: Génération FRACTAL ADL pour une application de type client-serveur.

section 5.1 en remplaçant tout le code technique des composants par des annotations. Ainsi, l'interface `Service` décrite dans le listing 5.8 est enrichie de l'annotation @provides afin de préciser que cette interface est une interface serveur FRACTAL et que son nom par défaut est `s` (ligne 1).

```
1 /** @provides name=s */
2 public interface Service {
3   void print(String message);
4 }
```

LST. 5.8: Description annotée de l'interface Service.

La classe `Client` décrite dans le listing 5.9 précise de la même façon que le composant Client fournit une interface `Runnable` nommée `r` (ligne 1)[3]. La méthode `init` marquée par l'annotation @lifecycle définit le code d'initialisation à exécuter lors du démarrage du composant (ligne 4). Le champ `s` est marqué par l'annotation @requires pour que la sémantique d'une interface requise soit automatiquement appliquée telle qu'elle est définie dans le modèle de composants FRACTAL (ligne 5). Enfin, le code métier du composant Client demeure identique (ligne 6).

```
1 /** @provides name=r signature=java.lang.Runnable */
2 public class Client implements Runnable {
3   protected String message;
4   /** @lifecycle when=start */
5   public void init() { this.message = "hello world"; }
6   /** @requires */
7   protected Service s;
8   public void run() { this.s.print(this.message); }
9 }
```

LST. 5.9: Mise en œuvre annotée du composant Client.

La classe `Server` décrite dans le listing 5.10 fournit l'interface `Service` (ligne 1). Le champ `logger` est marqué avec l'annotation @logger pour préciser que le champ fait référence à un outil de trace fourni par le substrat d'exécution de FRACTAL (ligne 2). Le champ `header` est marqué avec l'annotation @attribute pour que ce champ soit considéré comme un attribut du composant Server (ligne 3). Le code métier du composant Server ne nécessite aucune modification (ligne 4).

Finalement, le listing 5.11 présente le composant composite HelloWorld. Cette définition hérite de la définition abstraite de composant composite générée automatiquement pour le composant Client (ligne 1). Dès lors, il suffit uniquement de préciser la définition du composant fournissant

[3] Lorsqu'elle marque une classe, l'annotation @provides est utilisée pour marquer une interface héritée, elle-même marquée ou non.

```
1 public class Server implements Service {
2   /** @logger */
3   protected Logger logger;
4   /** @attribute */
5   protected String header;
6   public void print(String msg) { this.logger.info(this.header + msg); }
7 }
```

LST. 5.10: Mise en œuvre annotée du composant Server.

l'interface Service. La définition utilisée dans ce cas précis est celle du composant Server pour lequel il est nécessaire de préciser la valeur de l'attribut header.

```
1 <definition name="HelloWorld" extends="ClientComposite">
2   <component name="s" definition="Server(->)"/>
3 </definition>
```

LST. 5.11: Description FRACTAL ADL du composant HelloWorld.

Les générateurs décrits dans la section 5.3.2 se chargent ensuite de produire le code technique et les descriptions ADL rigoureusement conformes au modèle de composants FRACTAL. L'application HelloWorld décrite dans les listings 5.8 à 5.11 est donc sémantiquement équivalente à l'application présentée dans la section 5.1. En revanche, l'application a gagné en lisibilité et en concision. En effet, les composants ne mélangent plus les codes métier et technique sans distinction. De plus, la taille globale du code source (fichiers Java et FRACTAL ADL) est passée de 69 lignes à 23 lignes sans pénaliser le comportement de l'application (soit tout de même une réduction de 76% !).

5.4 Éléments d'implantation de FRACLET

Cette section présente quelques éléments liés à l'implantation de FRACLET. Nous présentons ici l'implantation de FRACLET utilisant les annotations de type XDoc et le moteur de génération XDOCLET [WR03]. FRACLET est également implanté et disponible dans une version utilisant les annotations Java5 et le moteur de transformation SPOON [Paw05]. FRACLET est disponible sous licence libre LGPL à l'adresse suivante : http://fractal.objectweb.org/tutorials/fraclet.

5.4.1 Description du moteur de génération

L'architecture du moteur de génération utilisé par FRACLET est présentée dans la figure 5.6. Ce moteur de génération repose sur XDOCLET pour réaliser la génération des fichiers Java et FRACTAL ADL. XDOCLET assure la liaison entre l'analyseur Java et les générateurs d'artefacts. Dans un premier temps, XDOCLET utilise la description des annotations FRACLET au format QTAGS pour en générer une représentation mémoire nommée annotations XDoc. Cette représentation mémoire est utilisée dans un second temps par l'analyseur Java pour charger le code source annoté de l'application sous la forme d'un modèle Java défini par la bibliothèque QDox[4]. Dans un troisième temps, chaque générateur d'artefact utilise le modèle Java pour produire les artefacts définis par FRACLET. Un générateur FRACLET se compose d'un Template et d'un Plugin. Le Plugin assure l'indépendance entre les annotations et les artefacts générés. En particulier, le Plugin accède aux annotations XDoc et au modèle Java pour alimenter le Template associé au générateur. Le Template correspond à un fichier source paramétré dont certaines parties sont remplacées lors du processus de génération. Les Template de type Java utilisent le

[4] Le projet QDox : http://qdox.codehaus.org/

moteur de génération VELOCITY[5] tandis que les Template de type XML reposent sur le moteur de génération JELLY[6]. Ce dernier est celui que nous utilisons pour générer et valider les descriptions FRACTAL ADL.

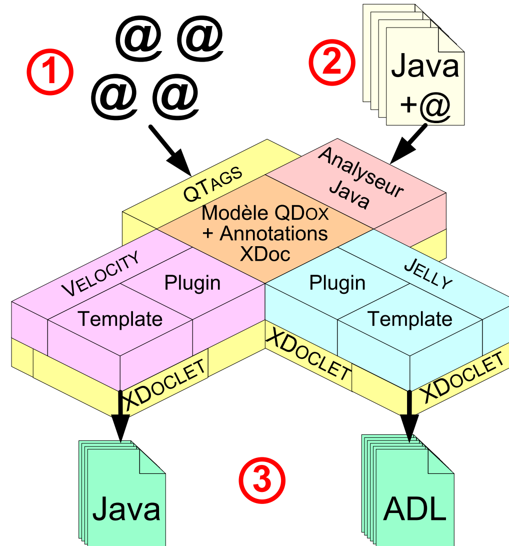


FIG. 5.6: Architecture du moteur de génération FRACLET.

5.4.2 Illustration du code généré

Cette section présente les différents artefacts générés par FRACLET pour l'application HelloWorld revisitée dans la section 5.3.3. Le listing 5.12 présente le résultat de la génération des interfaces de contrôle des attributs. Dans le cas de l'application HelloWorld, une interface `ServerAttributes` intégrant les méthodes `getHeader` et `setHeader` pour accéder et définir la valeur de l'attribut header du composant Server.

```
1 public interface ServerAttributes extends AttributeController {
2   String getHeader();
3   void setHeader(String header);
4 }
```

LST. 5.12: Code source généré pour l'attribut header.

Les listings 5.13 et 5.15 présentent le résultat de l'exécution du générateur de composants primitifs sur l'application HelloWorld. Le listing 5.13 correspond au composant généré à partir de la classe `Client` et intègre le contrôle du cycle de vie et des liaisons du composant.

Le listing 5.15 correspond au composant généré à partir de la classe `Server` et supporte la configuration de la trace et le contrôle de l'attribut header du composant.

Les listings 5.14, 5.16 et 5.17 illustrent le résultat de l'exécution du générateur de définitions primitives. La définition présentée dans le listing 5.14 est issue de l'analyse de l'interface `Service`. Celle-ci décrit un composant abstrait fournissant une interface de type `Service` dont le nom `s` a été défini par l'attribut *name* de l'annotation @provides.

La description FRACTAL ADL `Client` décrite dans le listing 5.16 est obtenue à partir de la classe `Client` de l'application HelloWorld. Cette description définit les interfaces fournies et

[5]Le moteur de génération VELOCITY : http://jakarta.apache.org/velocity/
[6]Le moteur de génération JELLY : http://jakarta.apache.org/commons/jelly/

```
1 public class FcClient extends Client implements LifeCycleController, BindingController {
2   // LifeCycleController interface
3   public String getFcState() { return null; }
4   public void startFc() { super.init(); }
5   public void stopFc() { }
6   // BindingController interface
7   public String[] listFc() {
8     ArrayList _itf_ = new ArrayList();
9     _itf_.add("s");
10    return (String[]) _itf_.toArray(new String[1]);
11  }
12  public Object lookupFc(String name) throws NoSuchInterfaceException {
13    if (name.equals("s")) return super.s;
14    throw new NoSuchInterfaceException(name+" is unknown!");
15  }
16  public void bindFc(String name, Object itf) throws NoSuchInterfaceException {
17    if (name.equals("s")) super.s = (Service) itf;
18    else throw new NoSuchInterfaceException(name+" is unknown!");
19  }
20  public void unbindFc(String name) throws NoSuchInterfaceException {
21    if (name.equals("s")) super.s = null;
22    else throw new NoSuchInterfaceException(name+" is unknown!");
23 } }
```

LST. 5.13: Code source généré pour le composant Client.

```
1 <definition name="Service">
2   <interface name="s" signature="Service" role="server" cardinality="singleton"/>
3 </definition>
```

LST. 5.14: Description FRACTAL ADL générée pour l'interface Service.

```
1 public class FcServer extends Server implements ServerAttributes, Loggable {
2   // Loggable interface
3   public Logger getLogger() { return super.logger; }
4   public void setLogger(Logger l) { super.logger = l; }
5   private LoggerFactory _factory_;
6   public Logger getLoggerFactory() { return _factory_; }
7   public void setLoggerFactory(LoggerFactory f) { _factory_ = f; }
8   // ServerAttributes interface
9   public String getHeader() { return super.header; }
10  public void setHeader(String header) { super.header = header; }
11 }
```

LST. 5.15: Code source généré pour le composant Server.

requises par le composant Client et précise que le contenu du composant Client correspond à la classe Java `FcClient` qui a été générée par FRACLET.

```
1<definition name="Client">
2  <interface name="r" signature="java.lang.Runnable" role="server" cardinality="singleton"/>
3  <interface name="s" signature="Service" role="client" cardinality="singleton"/>
4  <content class="FcClient"/>
5</definition>
```

LST. 5.16: Description FRACTAL ADL générée pour le composant Client.

La description FRACTAL ADL `Server` décrite dans le listing 5.17 est obtenue à partir de la classe `Server`. Cette description étend le contenu de la description `Service` en précisant la classe Java d'implantation du composant et la liste des attributs disponibles sur le composant Server. Dans la mesure où aucune valeur par défaut n'a été définie pour l'attribut header de ce composant, cet attribut est réifié comme un argument de la description `Server` afin que sa valeur puisse être définie *a posteriori*.

```
1<definition name="Server" extends="Service" arguments="header">
2  <content class="FcServer"/>
3  <attributes signature="ServerAttributes">
4    <attribute name="header" value="${header}"/>
5  </attributes>
6</definition>
```

LST. 5.17: Description FRACTAL ADL générée pour le composant Server.

Enfin, le listing 5.18 présente le résultat de l'exécution du générateur de définitions composites abstraites. Cette description correspond à un motif d'assemblage de type client-serveur généré pour tout composant disposant d'au moins une interface requise de type singleton et obligatoire. Pour l'application HelloWorld, ce générateur produit une description `ClientComposite` qui exporte l'interface `r` fournie par le composant Client et lie l'interface `s` requise par ce même composant à une interface `s` d'un composant abstrait nommé `s`.

```
1<definition name="ClientComposite">
2  <interface name="r" signature="Runnable" role="server" cardinality="singleton"/>
3  <component name="client" definition="Client"/>
4  <component name="s" definition="Service"/>
5  <binding client="this.r" server="client.r" />
6  <binding client="client.s" server="s.s" />
7</definition>
```

LST. 5.18: Description FRACTAL ADL générée pour le composite Client.

Discussion. Le code généré par FRACLET présenté dans cette section est sémantiquement équivalent au code que le développeur d'applications FRACTAL peut écrire manuellement. L'approche générative réduit non seulement la taille du code à développer mais offre également plus de fiabilité en systématisant le code technique et en éliminant ainsi tout risque d'erreur.

Cependant, cette approche présente quelques limites techniques. En particulier, notre approche de génération par héritage introduit une classe supplémentaire (préfixée par `Fc`) pour chaque composant développé. Cette classe supplémentaire se traduit par une augmentation de l'empreinte mémoire de l'application comparé à une approche de développement traditionnelle d'une application FRACTAL. De plus, les éléments de code annotés ne peuvent pas utiliser la visibilité **`private`** du langage Java car ils seront accédés depuis une classe fille.

Une solution possible pour résoudre ces limitations consiste à utiliser le moteur de transformation SPOON [Paw05] afin d'introduire le code technique du modèle de composants FRACTAL

directement dans le code source des composants primitifs de l'application. Cependant, cette solution repose sur l'utilisation des annotations Java 5 et limite les environnements cibles sur lesquels l'application peut être déployée. En particulier, l'exécution de l'application sur un environnement contraint dépend de la disponibilité d'un support d'exécution compatible avec Java 5.

5.5 Évaluations

Cette section présente des évaluations quantitatives du code source et du code généré avec FRACLET. Une évaluation qualitative de FRACLET est ensuite introduite. Cette évaluation est basée sur des constats liés à l'utilisation de FRACLET dans des développements logiciels utilisant le modèle de composants FRACTAL.

5.5.1 Évaluation quantitative du code source

Dans un premier temps, nous souhaitons évaluer la taille du code source d'une application développée avec le modèle de composants FRACTAL. Nous comparons une version complète de l'application développée sans utiliser FRACLET à une version utilisant le modèle de programmation FRACLET.

Le tableau 5.2 présente une évaluation de FRACLET réalisée sur l'exemple de l'application HelloWorld (présenté dans la section 5.1). Pour ce faire, nous utilisons le code de l'application tel qu'il est distribué par le projet FRACTAL. Les commentaires et les lignes vides ne sont pas pris en compte par cette évaluation. Cette évaluation montre que 49 % du code Java peut être économisé en utilisant FRACLET. Cette économie correspond non seulement aux interfaces de contrôle des attributs générées par FRACLET mais également au code technique des contrôleurs qu'il est nécessaire d'implanter. De la même façon, la génération automatique des descriptions FRACTAL ADL associées aux composants primitifs et des descriptions abstraites des composants composites permet d'économiser 77 % de la taille des fichiers de description FRACTAL ADL qu'il est nécessaire de maintenir lors du développement d'une application avec le modèle de composants FRACTAL.

Code applicatif	Unité	FRACTAL A	FRACLET B	Gain $G = A - B$	Taux G/A
Java	fichiers	4	3	1	25 %
ADL	fichiers	9	2	7	78 %
Java	lignes	70	36	34	**49 %**
ADL	lignes	75	17	58	**77 %**
Source	octets	6 K	2 K	4 K	67 %

TAB. 5.2: Mesures empiriques du code source réalisées sur l'exemple HelloWorld.

L'exemple de l'application HelloWorld présenté dans ce chapitre peut être considéré comme non représentatif d'une application réelle. Le tableau 5.3 présente une évaluation de FRACLET réalisée sur le serveur Web Comanche. Comanche présente la particularité de requérir peu des spécificités du modèle de composants FRACTAL (liaisons, attributs, etc.). Cette évaluation empirique vise à observer le bénéfice de FRACLET sur une application *a priori* peu sujette au problème de mélange du code métier et technique. Il apparait néanmoins que 28 % du code Java de Comanche peut être supprimé et remplacé par des annotations. Ces annotations permettent également de réduire de 66 % la taille des fichiers FRACTAL ADL. En économisant globalement 44 % du code source du serveur Web Comanche, nous montrons que les bénéfices de FRACLET peuvent s'appliquer à n'importe quelle application FRACTAL.

Code applicatif	Unité	FRACTAL A	FRACLET B	Gain $G = A - B$	Taux G/A
Java	fichiers	13	12	1	8 %
ADL	fichiers	19	6	13	68 %
Java	lignes	263	189	74	**28 %**
ADL	lignes	137	47	90	**66 %**
Source	octets	14 K	8 K	6 K	**44 %**

TAB. 5.3: Mesures empiriques du code source réalisées sur le serveur Web Comanche.

5.5.2 Évaluation quantitative du code généré

Dans cette section, nous observons la taille du code généré par FRACLET à partir d'un code source annoté. Nous montrons que la taille du code généré par FRACLET peut représenter jusqu'à 50 % du code de l'application finale.

Par exemple, l'évaluation du code généré de l'application HelloWorld présentée dans le tableau 5.4 montre que l'utilisation de FRACLET sur une application ne contenant que deux composants permet de générer jusqu'à 84 % du code applicatif.

Code applicatif	Unité	Code Source A	Code Généré B	Ratio $B/(A+B)$
Java	fichiers	3	3	50 %
ADL	fichiers	2	4	67 %
Java	lignes	36	203	85 %
ADL	lignes	17	29	63 %
Source	octets	2 K	10 K	**84 %**

TAB. 5.4: Mesures empiriques du code généré réalisées sur l'application HelloWorld.

L'évaluation du code généré pour serveur Web Comanche présentée dans le tableau 5.5 montre que l'utilisation de FRACLET s'avère pertinente pour le serveur Web Comanche même si peu de ses composants ont des dépendances vers les interfaces imposées par le modèle de composants FRACTAL. Notamment, FRACLET permet de générer 76 % du code et des descripteurs du serveur Web Commanche.

Code applicatif	Unité	Code Source A	Code Généré B	Ratio $B/(A+B)$
Java	fichiers	12	6	33 %
ADL	fichiers	6	14	70 %
Java	lignes	189	519	73 %
ADL	lignes	47	84	64 %
Source	octets	8 K	26 K	**76 %**

TAB. 5.5: Mesures empiriques du code généré réalisées sur le serveur Web Comanche.

Le tableau 5.6 présente l'évaluation du projet THREAD.MANAGEMENT FRAMEWORK (TMF). TMF est un canevas logiciel permettant de surveiller et de contrôler l'allocation des *threads* d'une application de manière transparente pour le développeur. L'implantation de ce canevas avec FRACLET présente la particularité de ne comporter que quatre composants primitifs spécifiques au modèle de composants FRACTAL dont l'assemblage est décrit par un seul composite. Les autres fichiers correspondent à des composants indépendants des interfaces du modèle FRACTAL qui justifient la génération des 24 descriptions d'architecture pour l'ensemble des composants du projet. Dans ce type de situation où peu de code technique FRACTAL nécessite d'être généré, FRACLET permet néanmoins de simplifier l'assemblage des composants de l'application en générant

93 % des descripteurs FRACTAL ADL de tous les composants de l'application (même ceux qui ne dépendent pas du modèle de composants FRACTAL).

Code applicatif	Unité	Code Source A	Code Généré B	Ratio $B/(A+B)$
Java	fichiers	26	4	13 %
ADL	fichiers	1	24	**96 %**
Java	lignes	2440	260	10 %
ADL	lignes	10	131	**93 %**
Source	octets	136 K	38 K	22 %

TAB. 5.6: Mesures empiriques du code généré réalisées sur le projet TMF.

Le tableau 5.7 détaille l'évaluation du code généré par FRACLET pour notre canevas intergiciel GOTM. En appliquant notre modèle de programmation FRACLET au canevas GOTM, nous avons pu réduire le nombre de composants en factorisant et en paramétrant certaines fonctionnalités. Par conséquent, GOTM utilise intensivement les caractéristiques du modèle de composants FRACTAL et cela se manifeste par un volume de code généré très proche de la taille du code source annoté (47 %).

Code applicatif	Unité	Code Source A	Code Généré B	Ratio $B/(A+B)$
Java	fichiers	63	40	39 %
ADL	fichiers	20	92	82 %
Java	lignes	2451	2977	55 %
ADL	lignes	391	538	58 %
Source	octets	200 K	178 K	**47 %**

TAB. 5.7: Mesures empiriques du code généré réalisées sur le projet GOTM.

Le tableau 5.8 présente l'évaluation pour le projet FRACTAL DEPLOYMENT FRAMEWORK (FDF). FDF est un canevas logiciel dédié au déploiement de systèmes distribués. FDF est un projet récent qui applique la démarche de construction à granularité extrêmement fine décrite dans cette thèse. L'utilisation de FRACLET sur un projet contenant un grand nombre de composants de très fine granularité permet de générer jusqu'à 41 % du code final de l'application. De plus, en appliquant immédiatement notre démarche, FDF a pu accroître rapidement la taille de sa bibliothèque au point de disposer d'environ deux fois plus de composants que le projet GOTM sur une période de développement deux fois plus courte.

Code applicatif	Unité	Code Source A	Code Généré B	Ratio $B/(A+B)$
Java	fichiers	126	79	39 %
ADL	fichiers	98	128	57 %
Java	lignes	3083	3673	54 %
ADL	lignes	1119	744	40 %
Source	octets	479 K	329 K	**41 %**

TAB. 5.8: Mesures empiriques du code généré réalisées sur le projet FDF.

5.5.3 Évaluation qualitative

FRACLET a été utilisé intensivement pour le développement de plusieurs canevas logiciels tels que GOTM [RSAM06b], FRACTAL DEPLOYMENT FRAMEWORK [FM06] et COSMOS [Con06].

Chacun de ces canevas est constitué de plusieurs dizaines voire centaines de composants primitifs. Au delà d'un simple gain quantitatif sur la taille du code de ces canevas, FRACLET a apporté différents gains qualitatifs pour la réalisation de ces canevas dont (1) la suppression des erreurs de programmation liées à FRACTAL, (2) la réduction des temps de développement et de mise au point, et enfin (3) l'indépendance du code des composants primitifs vis-à-vis des interfaces de programmation de FRACTAL. Ces gains ont permis aux concepteurs de ces canevas de se concentrer sur le métier à réaliser plutôt que de " perdre " du temps sur la manière d'implanter ce métier via le modèle de composants FRACTAL.

5.6 Généralisation à d'autres modèles de composants

Cette section présente une généralisation du modèle de programmation FRACLET pour adresser la programmation par composants. L'objectif de cette généralisation est de fournir une abstraction des modèles de programmation existants. Différents générateurs permettent alors d'introduire les modèles de programmation de différents modèles de composants. Ainsi, les générateurs pour le modèle de composants OPENCOM sont détaillés dans la suite de cette section. Grâce à cette extension, il est possible d'exécuter une même application avec différents modèles de composants sans modifier son code métier.

5.6.1 Programmation orientée composant et dirigée par le code

Notre proposition vise à appliquer les principes de l'ingénierie dirigée par les modèles (MDE) (en anglais, *Model-Driven Engineering*) au niveau du code des composants de l'application. Notre objectif est de trouver un formalisme permettant d'exprimer le code métier des composants de l'application sans créer de dépendance directe vers un modèle de composants particulier. Nous définissons ainsi le terme de PIC (en anglais, *Platform-Independent Component*) pour désigner un code applicatif indépendant des modèles de composants existants. Nous employons le terme de PSC (en anglais, *Platfom-Specific Component*) pour désigner un code applicatif dédié à un modèle de composants particulier (par exemple, FRACTAL, OPENCOM). Le passage de PIC à PSC est assuré par des générateurs de code qui enrichissent le code applicatif des composants avec le modèle de programmation d'un modèle de composants particulier comme illustré dans la figure 5.7.

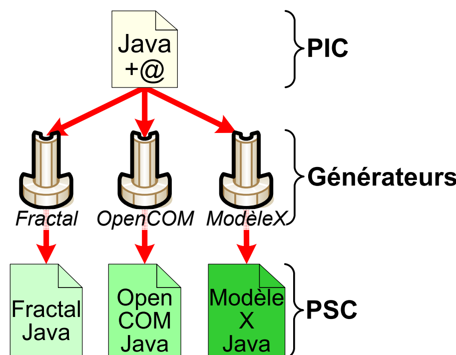


FIG. 5.7: Illustration des générateurs de modèles de programmation spécifiques.

En pratique, le PIC est un composant applicatif marqué avec des annotations liées au domaine de la programmation par composants. Ces annotations intègrent des informations spécifiques aux concepts de composants et qui ne sont pas exprimées dans le code de l'application. Le jeu d'annotation que nous utilisons a été présenté dans la section 5.3.1. Chaque générateur est dédié à un modèle de composants particulier. Celui-ci interprète le code annoté et le complète avec les

propriétés non-fonctionnelles supportées par le modèle de composants (par exemple, gestion des liaisons, du cycle de vie, des attributs).

5.6.2 Description des générateurs OPENCOM

OPENCOM (présenté dans la section 3.3.2 du chapitre 3) propose une implantation en Java de son modèle de programmation [CBG+04, Nex05]. Ce modèle de programmation introduit un certain nombre d'interfaces spécifiques. Parmi ces interfaces, les interfaces `IUnknown`, `IMetaInterface`, `ILifeCycle`, et `IConnections` sont nécessaires au développement d'un composant OPENCOM. Les interfaces `IUnknown`, `IMetaInterface` et `ILifeCycle` décrivent les interfaces disponibles, le contrôle du composant et la gestion du cycle de vie, respectivement. L'interface `IConnections` est disponible sur les composants disposant d'au moins un réceptacle (ou interface requise).

Afin d'introduire le code technique nécessaire à l'implantation de ces quatre interfaces, un seul générateur est nécessaire pour OPENCOM.

Composant primitif est donc le générateur de code Java qui pour chaque composant annoté, produit le code technique requis par le modèle de programmation OPENCOM.

5.6.3 Illustration du code généré pour OPENCOM

Cette section reprend l'exemple de l'application HelloWorld revisitée dans la section 5.3.3 afin de pouvoir exécuter le code annoté avec le modèle de composants OPENCOM. L'architecture de cette application en considérant le modèle de composants OPENCOM est décrite dans la figure 5.8. Cette architecture repose sur deux composants Client et Server connectés entre eux.

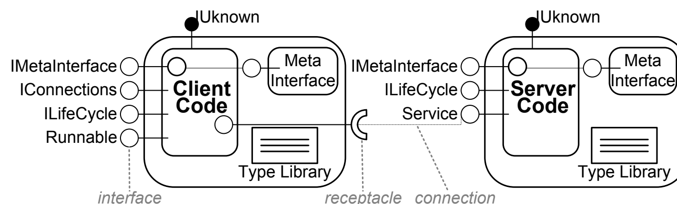


FIG. 5.8: Description des composants OPENCOM HelloWorld.

Le listing 5.19 présente à titre d'exemple le code technique OPENCOM produit pour le composant Client. Ce code technique inclut l'implantation des méthodes requises par les interfaces `IUnknown`, `IMetaInterface`, `ILifeCycle`, et `IConnections` ainsi que la déclaration des méta-données associées.

Le listing 5.20 présente le code technique OPENCOM produit pour le composant Server. Ce code technique inclut l'implantation des méthodes requises par les interfaces `IUnknown`, `IMetaInterface`, et `ILifeCycle` ainsi que la déclaration des méta-données associées. Ce listing illustre la prise en charge automatique de l'attribut `header` du composant.

5.6.4 Évaluation

Le bénéfice de cette proposition ne se limite plus à la réduction de la taille du code source et du nombre d'erreurs liées à l'implantation du code technique. En effet, le gain que nous observons ici réside dans la possibilité d'exécuter un même code métier annoté sur différents substrats d'exécution (FRACTAL, OPENCOM, etc.) sans modifier ce code métier. Le code métier est donc isolé des préoccupations techniques liées au modèle de composants utilisé pour réaliser l'application et le protège ainsi des évolutions possibles des modèles de programmation. Alors que la modification d'une interface du modèle de programmation impacte toutes les applications

```
1 public class OCMClient extends Client
2 implements IUnknown, IMetaInterface, IConnections, ILifeCycle {
3   private OCM_SingleReceptacle ocm_s = new OCM_SingleReceptacle(Service.class);
4   private MetaInterface _meta_;

6   public OCMClient(IUnknown binder) {
7     OCM_SingleReceptacle ocm = new OCM_SingleReceptacle(IOpenCOM.class);
8     ocm.connectToRecp(binder,"OpenCOM.IOpenCOM", 0);
9     _meta_ = new MetaInterface((IOpenCOM) ocm.m_pIntf, this);
10  }
11  // IUnknown Interface
12  public Object QueryInterface(String name) {
13    if (name.equalsIgnoreCase("Runnable"))
14      return this;
15    Vector query = new Vector();
16    _meta_.ReadInterfaceNames(this.getClass(), query);
17    for (int i=0; i<query.size(); i++)
18      if (name.equalsIgnoreCase(query.get(i).toString()))
19        return this;
20    return null;
21  }
22  // IMetaInterface Interface
23  public int enumIntfs(Vector ppIntf) {
24    return _meta_.enumIntfs(this,ppIntf);
25  }
26  public int enumRecps(Vector recp) {
27    return _meta_.enumRecps((IUnknown)this,recp);
28  }
29  public boolean SetAttributeValue(String id, String kind, String name,
30                                   String type, Object val) {
31    return _meta_.SetAttributeValue(id, kind, name, type, val);
32  }
33  public TypedAttribute GetAttributeValue(String id, String kind, String name) {
34    return _meta_.GetAttributeValue(id, kind, name);
35  }
36  public Hashtable GetAllValues(String kind, String id) {
37    return _meta_.GetAllValues(kind, id);
38  }
39  // ILifeCycle Interface
40  public boolean startup(Object pIOCM) {
41    super.init();
42    return true;
43  }
44  public boolean shutdown() {
45    return true;
46  }
47  // IConnections Interface
48  public boolean connect(IUnknown itf, String signature, long id) {
49    boolean r = ocm_s.connectToRecp(itf, signature, id);
50    if (r && signature.equalsIgnoreCase("Service"))
51      super.s = (Service)ocm_s.m_pIntf;
52    return r;
53  }
54  public boolean disconnect(String signature, long id) {
55    boolean r = ocm_s.disconnectFromRecp(id);
56    if (r && signature.equalsIgnoreCase("Service"))
57      super.s = null;
58    return r;
59 } }
```

LST. 5.19: Composant OPENCOM Client.

```
1 public class OCMServer extends Server implements IUnknown, IMetaInterface, ILifeCycle {
2   private MetaInterface _meta_;
3   public OCMServer(IUnknown binder) {
4     OCM_SingleReceptacle ocm = new OCM_SingleReceptacle(IOpenCOM.class);
5     ocm.connectToRecp(binder, "OpenCOM.IOpenCOM", 0);
6     _meta_ = new MetaInterface((IOpenCOM) ocm.m_pIntf, this);
7     super.logger = JavaLog.logger("Server");
8   }
9   // IUnknown Interface
10  public Object QueryInterface(String name) {
11    if (name.equalsIgnoreCase("Service")) return this;
12    Vector query = new Vector();
13    _meta_.ReadInterfaceNames(this.getClass(), query);
14    for (int i=0; i<query.size(); i++)
15      if (name.equalsIgnoreCase(query.get(i).toString())) return this;
16    return null;
17  }
18  // IMetaInterface Interface
19  public int enumIntfs(Vector ppIntf) {
20    return _meta_.enumIntfs((IUnknown) this, ppIntf);
21  }
22  public int enumRecps(Vector recp) {
23    return _meta_.enumRecps((IUnknown) this, recp);
24  }
25  public boolean SetAttributeValue(String id, String kind, String name,
26                                    String type, Object val) {
27    if (name.equalsIgnoreCase("header")) super.header = (String) val;
28    return _meta_.SetAttributeValue(id, kind, name, type, val);
29  }
30  public TypedAttribute GetAttributeValue(String id, String kind, String name) {
31    if (name.equalsIgnoreCase("header")) return super.header;
32    return _meta_.GetAttributeValue(id, kind, name);
33  }
34  public Hashtable GetAllValues(String kind, String id) {
35    return _meta_.GetAllValues(kind, id);
36  }
37  // ILifeCycle Interface
38  public boolean startup(Object pIOCM) { return true; }
39  public boolean shutdown() { return true; }
40 }
```

LST. 5.20: Composant OPENCOM Server.

développées traditionnellement avec ce modèle, notre proposition n'impacte que le générateur requiérant cette interface.

5.7 Travaux connexes

Cette section compare notre proposition avec les travaux existants dans le domaine de la programmation par attributs (*cf.* section 5.7.1), de la programmation générative (*cf.* section 5.7.2), de la programmation par aspects (*cf.* section 5.7.3) et des langages dédiés (*cf.* section 5.7.4)

5.7.1 Programmation par attributs

La programmation par attributs fait l'objet d'un intérêt croissant dans le domaine de la programmation par composants. Récemment, la spécification Enterprise Java Bean (EJB) 3.0 a intégré les annotations afin de simplifier le développement de composants EJB [DK05]. De façon similaire, la spécification Service Component Architecture (SCA) intègre également les annotations dans son modèle de programmation Java [IBM05]. Néanmoins, seules les annotations sont décrites dans ces spécifications et les outils de génération du code technique associés restent la charge du développeur des serveurs d'application de type EJB ou SCA. Dans ce chapitre, nous présentons non seulement un jeu d'annotations pour le modèle de composants Fractal mais nous identifions également les différents générateurs nécessaires pour FRACTAL et nous présentons une généralisation de FRACLET pour le support d'autres modèles de composants tels que OPENCOM.

La programmation par attributs a déjà été appliquée dans FRACTAL dans une approche *a posteriori* [SQ06]. Il s'agit de compléter FRACTAL ADL afin de pouvoir marquer les interfaces d'une application distribuée pour la rendre plus résistante aux attaques de type *déni de service*. Cette proposition ne requiert aucune modification du code source de l'application puisqu'elle intervient au niveau de FRACTAL ADL et peut s'appliquer par conséquent lors du déploiement de l'application. Notre proposition utilise également intensivement des annotations mais vise la simplification du développement d'applications à base de composants FRACTAL. Nos annotations s'appliquent dans notre cas au code source de l'application et adressent la génération du code technique imposé par le modèle de programmation de FRACTAL. De plus, notre proposition ne requiert aucune modification des outils existants mais elle les complète de façon transparente.

5.7.2 Programmation générative

L'ingénierie dirigée par les modèles (MDE) propose d'adresser le problème de la complexité du code des applications en utilisant des modèles de haut niveau pour décrire les fonctionnalités de l'application [SVV06]. À partir de ces modèles de haut niveau, le code de l'application peut être automatiquement produit pour un modèle de programmation particulier grâce à des mécanismes de transformation et de génération. Néanmoins, le code généré intègre aussi bien les préoccupations métier liées à l'application que les préoccupations techniques liées au modèle de programmation. Notre proposition ne concurrence pas l'approche dirigée par les modèles mais propose une solution pour séparer les préoccupations techniques des préoccupations métier d'une application. Dans ce contexte, une approche MDE pourrait se limiter à générer un code applicatif annoté. Ce code annoté est ensuite interprété par FRACLET pour produire une version de l'application exécutable avec un modèle de programmation particulier. Ainsi, le processus MDE ne nécessite pas de prendre en compte les différents modèles de composants disponibles sur le marché mais il utilise le support fourni par FRACLET. Le processus MDE est ainsi simplifié et son évolution n'est plus dépendante des modèles de programmation disponibles. Dans [WS05], les auteurs proposent notamment de transformer les stéréotypes UML appliqués dans les diagrammes UML en des annotations marquant le code applicatif généré. Ce code applicatif annoté est ensuite complété par des générateurs consommant les annotations.

La simplification du développement des composants FRACTAL a été adressée au travers de plusieurs outils. Par exemple, l'outil FRACTAL GUI permet de développer des composants FRACTAL en manipulant une représentation graphique de ces composants. Le code source et les descriptions FRACTAL ADL associés peuvent être générés à tout moment à partir de FRACTAL GUI. Néanmoins, le bémol de cette approche est que FRACTAL GUI ajoute des informations relatives à l'éditeur graphique dans les descriptions FRACTAL ADL. De plus, toute modification du code source de l'application ne peut être répercutée dans FRACTAL GUI car cet outil ne supporte pas le chargement d'une application existante. Notre approche n'introduit aucune information additionnelle dans les artefacts de FRACTAL. Les annotations servent à abstraire le code technique des composants et les générateurs permettent de produire le composant FRACTAL équivalent.

5.7.3 Programmation par aspects

La mise en œuvre AOKELL de la spécification FRACTAL, utilisant le paradigme de programmation par aspects, propose une solution pour la prise en compte de certaines propriétés techniques de FRACTAL [SPDC06]. Ainsi, cette solution consiste à intégrer automatiquement le code technique lié à l'interface de contrôle `BindingController` via l'utilisation d'un aspect configurable. Cet aspect utilise l'injection de code pour ajouter le code technique. Cependant, ce mécanisme ne permet pas la génération d'artefacts additionnels tels que l'interface de contrôle d'attributs. Notre approche permet non seulement d'introduire du code technique dans le composant mais elle permet également de générer des interfaces additionnelles, des descriptions FRACTAL ADL et des fichiers de configuration. De plus, notre approche n'est liée à aucune réalisation particulière du modèle de composants FRACTAL. Par conséquent, les composants générés avec FRACLET peuvent être exécutés sur n'importe quel substrat d'exécution FRACTAL tel que JULIA [BCL+06], AOKELL [SPDC06] ou PROACTIVE [BBC+06].

5.7.4 Langages dédiés

L'approche par langage dédié propose de définir un modèle de programmation spécifique à un domaine ou à un champ d'applications [vKV00]. Par exemple, dans le domaine des composants, le modèle de programmation ARCHJAVA (présenté dans la section 3.3.4) peut être considéré comme un langage dédié à la programmation des composants [ACN02b]. L'utilisation d'un langage dédié permet de définir une syntaxe sémantiquement plus riche que les langages de programmation généralistes. Par conséquent, grâce à cette précision sémantique, il est possible de réaliser une analyse des programmes écrits avec le langage dédié afin de vérifier que ceux-ci respectent bien les règles définies par les experts du domaine.

Cependant, il est reconnu que la conception, la mise en œuvre et la maintenance d'un langage dédié (et de son compilateur) correspondent à des opérations assez coûteuses et fastidieuses. En particulier, l'identification du domaine cible, c.-à-d. les abstractions caractéristiques est une opération délicate et stratégique pour la pertinence du langage dédié. De plus, l'équilibre entre le domaine cible et les constructions classiques des langages de programmation généralistes est souvent difficile à trouver.

FRACLET peut être également considéré comme un langage dédié au domaine de la programmation des composants. Les abstractions caractéristiques de la programmation par composants sont identifiées sous la forme d'un jeu d'annotations déterminé qui étend les constructions du langage de programmation généraliste et l'adapte ainsi à un domaine particulier. L'utilisation de ces annotations permet de réaliser un certain nombre de vérifications précoces spécifiques au domaine de la programmation par composants (chemins de communication, contrôle des reconfigurations, etc.). Le code associé aux annotations est transformé vers les constructions classiques du langage de programmation généraliste pour être exécuté par le substrat d'exécution d'un modèle de composant donné. Comparé à l'approche par langage dédié, l'approche par annotations fournit une définition plus modulaire d'un domaine. Il est ainsi possible d'écrire des programmes composant les annotations propres à plusieurs domaines (par exemple, composants, aspects) et de vérifier puis générer le code exécutable associé à ce programme. Le support des

annotations par le langage de programmation généraliste facilite la compréhension du modèle de programmation et permet de bénéficier des outils associés (compilateur, environnement de développement, outils de documentation, etc.).

Convergence des approches. FRACLET peut également être considéré comme un langage pivôt pour la construction de modèles de programmation de plus haut niveau. Ainsi, il est envisageable de construire à partir de FRACLET des représentations utilisant une abstraction de plus haut niveau telles qu'un langage dédié, un méta-modèle ou un profile UML. Dans cette optique, les annotations identifiées par FRACLET seraient converties en mot-clés dans un langage dédié, ou en stéréotypes dans le cadre de la définition d'un profil UML. Le processus de transformation de ces langages de haut niveau vers le langage générique intégrerait alors les méta-informations sous la forme d'annotations afin que celles-ci persistent dans le langage de programmation final.

5.8 Conclusion

Ce chapitre a présenté FRACLET, notre modèle de programmation pour faciliter le développement de composants FRACTAL. Ce modèle de programmation exploite les possibilités offertes par la programmation par attributs pour abstraire sous la forme d'annotations le code technique requis par le modèle de programmation FRACTAL. Cette proposition répond aux problèmes de mélange de code technique et métier ainsi qu'à la redondance des méta-informations présentes dans les descriptions d'architectures. En réduisant la taille du code métier des applications développées avec le modèle de composants FRACTAL, le code des applications gagne en lisibilité et les erreurs liées au code technique sont maîtrisées.

Le jeu d'annotations proposé par FRACLET est suffisamment généraliste pour qu'il puisse être utilisé avec d'autres modèles de composants. Nous avons ainsi présenté un générateur pour le modèle de composants OPENCOM afin de valider cette généralisation. De plus, avec FRACLET, il est désormais possible de maîtriser l'évolution des modèles de programmation et leur impact sur les applications existantes en n'adaptant que les générateurs. Dès lors, toute application annotée devient potentiellement exécutable avec différents modèles de composants (FRACTAL, OPENCOM, etc.).

Outre GOTM, FRACLET est également utilisé dans des projets de développement utilisant le modèle de composants FRACTAL tels que PROACTIVE [BBC+06], COSMOS [Con06], DACAR [DM06], FIESTA [WLD06], FRACTAL DEPLOYMENT FRAMEWORK [FM06], FRACTAL ASPECT COMPONENTS [PSCD06] ou THREAD MANAGEMENT FRAMEWORK[7].

Les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Nicolas Pessemier et ont donné lieu à plusieurs publications d'audience nationale et internationale [RPPM06b, RM06a, RPPM06a]. La version 2.0 de FRACLET est disponible sous licence libre LGPL à l'adresse suivante : `http://fractal.objectweb.org/tutorials/fraclet`.

Les perspectives associées à FRACLET sont nombreuses. Outre le développement de générateurs supplémentaires pour la prise en compte d'autres modèles de composants, nous souhaitons compléter l'analyse du code annoté des composants pour fiabiliser le développement des applications à base de composants. Par exemple, nous souhaitons intégrer une vérification des chemins de communications [LCL06] similaire à celui qui est disponible dans le modèle de composants ARCHJAVA [ACN02b]. Nous souhaitons également intégrer les travaux réalisés autour du système de typage des composants DREAM [BLQ+05] via la définition d'une annotation de méthode @dream.type permettant de décrire les conditions associées aux messages entrants et sortants des composants DREAM afin de faibiliser et d'inférer les types des assemblages des composants DREAM.

FRACLET offre ainsi une réponse efficace à la problématique de la programmation des abstractions fonctionnelles du canevas logiciel GOTM. Une fois développées, les abstractions fonc-

[7] Le projet THREAD MANAGEMENT FRAMEWORK : `http://tmf.gforge.inria.fr`.

tionnelles doivent être composées pour fournir une fonctionnalité concrète de l'intergiciel. Cette composition est l'objet du chapitre suivant.

The chances of a program doing what it's supposed to do is inversely proportional to the number of lines of code used to write it.

Murphy's Law

Chapitre 6

Le langage de description et de vérification de motifs d'architecture de GoTM

Sommaire

6.1 Introduction 100

6.2 Motivations 101

6.2.1 Prise en compte des constructions récurrentes 101

6.2.2 Séparation des préoccupations 102

6.3 Langage de description et de vérification de motifs d'architecture 103

6.3.1 Définition des invariants d'architecture 103

6.3.2 FPath : une syntaxe pour la navigation architecturale 104

6.3.3 Foreach : génération de motifs architecturaux pour Fractal ADL 106

6.3.4 Assert : vérification d'invariants architecturaux pour Fractal ADL 107

6.4 Illustration de l'utilisation des invariants architecturaux 108

6.4.1 Application à la définition du composant MySequence 108

6.4.2 Définition de motifs d'architecture génériques 110

6.5 Éléments d'implantation 113

6.5.1 Implantation de l'interpréteur FPath 113

6.5.2 Extension de l'usine Fractal ADL 116

6.5.3 Extension du contrôle des composants Fractal 118

6.5.4 Extension des générateurs Fraclet 119

6.5.5 Discussion 120

6.6 Travaux connexes 120

6.6.1 Description de motifs d'architecture 121

6.6.2 Vérification d'invariants d'architecture 121

6.6.3 Intégration de nouvelles préoccupations 121

6.7 Conclusion 122

CE CHAPITRE PRÉSENTE LE LANGAGE DE DESCRIPTION ET DE VÉRIFICATION de motifs d'architecture pour faire face à la complexité des descriptions d'architectures actuelles. En particulier, la multiplication des composants dans le développement des applications fait apparaître des constructions récurrentes dans les assemblages de composants. Ces constructions ont tendance à favoriser l'augmentation de la taille des descriptions d'architecture et des erreurs syntaxiques et sémantiques qui peuvent en découler. Or, la plupart de ces constructions récurrentes représentent des motifs d'assemblage dont l'application peut être automatisée sous certaines conditions. De plus, la complexité grandissante des descriptions s'accompagne également d'une augmentation des préoccupations prises en compte dans les descriptions. Celles-ci sont généralement mélangées dans un même descripteur.

Afin d'améliorer le passage à l'échelle des descriptions d'architecture et de réduire les erreurs syntaxiques et sémantiques, nous proposons d'intégrer deux nouveaux opérateurs de factorisation et de vérification de motifs d'architecture au langage FRACTAL ADL. *Ces opérateurs utilisent le langage* FPATH *pour exprimer des requêtes d'interrogation sur le contenu de l'architecture. Le résultat de ces requêtes est ensuite exploité par les opérateurs pour générer les motifs d'architecture ou pour détecter des erreurs d'assemblage. La détection de ces erreurs d'assemblage est également réalisée lors des reconfigurations dynamiques de l'architecture par une extension du niveau de contrôle des composants.*

6.1 Introduction

L'utilisation de FRACLET et des générateurs de descriptions FRACTAL ADL (présentés dans le chapitre précédent) permet de réduire drastiquement le nombre et la taille des descriptions FRACTAL ADL qu'il est nécessaire de décrire pour décrire une application à base de composants. Grâce à cet outil, l'architecte se contente désormais de définir les assemblages applicatifs en composant les descriptions FRACTAL ADL, qu'elles soient générées ou non. Dès lors, la décomposition d'une application en un grand nombre de composants est facilitée par la simplication du développement de ces composants. Cependant, cette composition peut requérir dans certains cas des tâches répétitives qui se manifestent par une augmentation de la taille de la description et des erreurs humaines liées à ces répétitions. Ces inconvénients impactent non seulement le temps de développement mais aussi la qualité de l'application.

Le contenu de ces descriptions intègre des préoccupations liées au métier de l'application décrite par l'architecte mais également des préoccupations liées à la technologie utilisée pour développer l'application. Ces préoccupations techniques peuvent identifier des invariants de l'architecture dont la sémantique est plus liée aux choix de conception qu'à l'expression du métier de l'application.

Pour remédier à ce problème, nous proposons de mettre en place deux nouveaux opérateurs de factorisation et de vérification du contenu des descriptions FRACTAL ADL. L'opérateur de factorisation vise à synthétiser les éléments répétitifs contenus dans les descriptions FRACTAL ADL afin d'alléger celles-ci et de réduire les erreurs rencontrées dans ces descriptions. Cet opérateur se présente sous la forme d'un nouveau mot-clé de l'ADL appelé `foreach` qui, en exploitant la syntaxe FPATH [Dav05], permet de capturer les motifs d'assemblages répétitifs sous une forme canonique. L'opérateur de vérification des descriptions FRACTAL ADL, appelé `assert`, utilise également la syntaxe FPATH pour identifier des invariants d'architecture. Ces invariants sont vérifiés sur l'architecture lors du chargement de la définition FRACTAL ADL et lors de la reconfiguration dynamique des architecture en cours d'exécution. Ainsi, la connaissance de ces invariants permet de vérifier que toute reconfiguration *ad-hoc* du composant respecte les invariants d'architecture définis par l'architecte.

La suite de ce chapitre est organisée de la manière suivante. La section 6.2 présente les constructions récurrentes et la séparation des préoccupations comme les motivations de notre contribution. La section 6.3 introduit nos deux nouveaux opérateurs du langage FRACTAL ADL

dédiés à la factorisation et à la vérification des motifs d'architecture. La section 6.4 illustre l'utilisation de ces opérateurs sur un exemple d'architecture et fournit ensuite différentes définitions génériques de motifs d'architecture. La section 6.5 présente différents éléments d'implantation liés à la réalisation du langage FPATH, à l'extension de l'usine FRACTAL ADL et à l'extension du contrôle des composants FRACTAL. La section 6.6 discute les travaux connexes à notre contribution et la section 6.7 conclut ce chapitre.

6.2 Motivations

Cette section expose les motivations pour un langage de description et de vérification de motifs d'architecture. Les problèmes que nous avons identifiés adressent la récurrence de certaines constructions dans les descripteurs ADL (*cf.* section 6.2.1) et le manque de séparation des préoccupations au sein de ces descripteurs (*cf.* section 6.2.2).

6.2.1 Prise en compte des constructions récurrentes

La multiplication des composants dans les intergiciels actuels fait apparaître de plus en plus de motifs d'assemblages au niveau de l'architecture. Ces motifs, qui étaient décrits statiquement dans le code des composants, sont désormais réifiés architecturalement dans les descripteurs d'assemblage. Cette réification architecturale implique par conséquent une augmentation de la taille et de la complexité des descriptions ADL et des erreurs qui peuvent en découler.

La figure 6.1 présente un exemple d'architecture avec différentes contructions récurrentes. Le composant MySequence représente une séquence d'étapes à réaliser sur un système d'information. Le système d'information est représenté par le composant SI. Ce composant fournit différentes interfaces liées au métier de l'application (interfaces *execute* et *configure*) et certaines d'entre elles peuvent être utilisées par l'ordonnanceur. Le composant SI requiert également une interface *logger* afin de tracer son exécution. Cette interface est connectée à l'interface *log* du composant Logger. Le composant MySequence définit également plusieurs composants Step. Chacun de ces composants représente une étape à réaliser sur le système d'information. Par conséquent ces composants utilisent également le composant SI à leur niveau. Étant donné que l'ensemble des étapes à exécuter doivent être réalisées sur le même système d'information, les composants SI locaux aux différentes étapes doivent être partagés avec le composant SI local à la séquence. Chaque étape peut également requérir l'accès au composant Logger en définissant une interface cliente *logger*. Enfin, la séquence des étapes ne peut se faire que si les interfaces *runnable* des différentes étapes sont connectées au composant Sequence Strategy qui gère l'ordonnancement des étapes à réaliser.

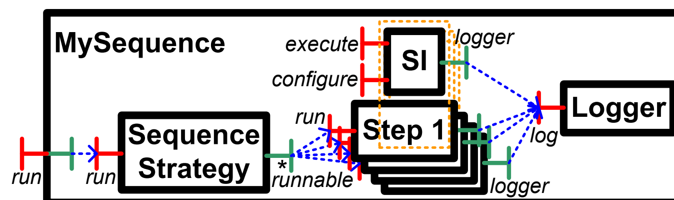


FIG. 6.1: Représentation graphique du composant Sequence.

Le listing 6.1 présente la définition FRACTAL ADL associée au composant MySequence. Cette définition reprend chaque élément caractéristique du composant MySequence pour composer l'ordonnanceur. Dans cette définition, seuls les composants `step1`, `step2` et `step3` requièrent d'accéder au système d'information (lignes 7–15). Les composants `si`, `step1`, `step2` et `step4` nécessitent l'utilisation d'un composant `Logger` pour tracer l'exécution des modifications réalisées sur le système d'information (lignes 23–26). Enfin, toutes les étapes sont connectées

au composant runner (lignes 19–22). Ce composant définit la stratégie utilisée pour exécuter la séquence d'étapes sur le système d'information (ligne 4).

```
1 <definition name="MySequence">
2   <interface name="run" role="server" signature="Runnable"/>

4   <component name="runner" definition="SequenceStrategy"/>
5   <component name="si"     definition="SI"/>
6   <component name="logger" definition="Logger"/>
7   <component name="step1" definition="Step1">
8     <component name="si" definition="./si"/>
9   </component>
10  <component name="step2" definition="Step2">
11    <component name="si" definition="./si"/>
12  </component>
13  <component name="step3" definition="Step3">
14    <component name="si" definition="./si"/>
15  </component>
16  <component name="step4" definition="Step4"/>

18  <binding client="this.run"              server="runner.run"/>
19  <binding client="runner.runnable-step1" server="step1.run"/>
20  <binding client="runner.runnable-step2" server="step2.run"/>
21  <binding client="runner.runnable-step3" server="step3.run"/>
22  <binding client="runner.runnable-step4" server="step4.run"/>
23  <binding client="si.logger"             server="logger.log"/>
24  <binding client="step1.logger"          server="logger.log"/>
25  <binding client="step2.logger"          server="logger.log"/>
26  <binding client="step4.logger"          server="logger.log"/>
27 </definition>
```

LST. 6.1: Description FRACTAL ADL du composant MySequence.

La définition du composant MySequence fait apparaître trois constructions récurrentes. La première construction consiste à connecter toutes les étapes au composant runner afin que celles-ci soient prises en compte dans la séquence. La seconde construction est la liaison des composants définissant une interface cliente logger au composant Logger. Enfin, la dernière construction récurrente est le partage du système d'information (représenté par le composant SI) avec les étapes qui s'appliquent sur celui-ci. Au-delà de la verbosité de cette définition, il apparait que ces constructions peuvent être soumises à certaines exceptions. En particulier, le composant step3 ne requiert pas l'utilisation d'un composant Logger alors que le composant step4 n'accède pas au système d'information. De plus, l'ajout d'une nouvelle étape implique la déclaration de l'instance du composant associé et de ses liaisons, et éventuellement le partage du composant si. Il est nécessaire de faire attention à ne pas systématiser certaines constructions dans des cas d'incompatibilité structurelle. Ce type de systématisation peut être la source de nombreuses erreurs dans les descriptions d'architectures.

6.2.2 Séparation des préoccupations

Un autre aspect auquel nous nous intéressons dans les langages de description d'architectures est la prise en compte de la séparation des préoccupations. En effet, la description d'une architecture fait intervenir de nombreuses préoccupations qui sont la plupart du temps mélangées dans une même définition. Par exemple, ces préoccupations peuvent être liées au métier de l'application ou aux choix techniques liés au développement de cette application. Ces choix techniques représentent des constructions à appliquer automatiquement aux éléments de la définition sous certaines conditions. Ces constructions ne sont pas issues directement de l'analyse du métier de l'application mais sont nécessaires à son bon fonctionnement.

Un exemple de préoccupation technique est la définition et la liaison du composant logger dans le listing 6.1. En effet, l'utilisation d'un composant de trace est lié au développement des composants métiers et leur liaison est réalisable à partir du moment où le composant métier dispose d'une interface cliente logger. De façon similaire, la définition du composant runner et des liaisons systématiques aux étapes peut apparaître comme une préoccupation technique liée

à l'utilisation du composant `runner`. Cette préoccupation technique stipule que les composants disposant d'une interface serveur `run` peuvent être connectés au composant `runner` afin de les appeler automatiquement lorsque le composant `MySequence` est invoqué.

La construction partageant le composant `SI` avec les étapes est plus proche d'une préoccupation métier liée au système d'information. Ce type de construction peut être apparenté à un invariant d'architecture précisant qu'une seule instance de composant `SI` est utilisée par le composant `MySequence`. Il apparaît que l'utilisation d'un composant technique ou métier peut s'accompagner d'un certain nombre de règles liées à son intégration. Ces règles peuvent être connues *a priori* (lors du développement du composant) ou définies *a posteriori* (lors de l'intégration du composant). Néanmoins, ces règles sont rarement explicitées et isolées car, la plupart du temps, elles sont appliquées et mélangées au reste de la définition.

Il apparaît donc intéressant de mettre en place un mécanisme de capitalisation des constructions récurrentes d'une description ADL. Ce mécanisme doit prendre en compte les aspects de séparation des préoccupations et permettre ainsi à l'architecte de l'application d'isoler les préoccupations techniques des préoccupations métiers lors de la définition de l'architecture. Il est nécessaire également de pouvoir vérifier la validité d'une intégration. Cette vérification assure que l'intégration d'une construction récurrente ne produit pas une architecture finale incohérente.

6.3 Langage de description et de vérification de motifs d'architecture

Cette section présente nos deux nouvelles constructions pour le langage de description d'architectures FRACTAL ADL favorisant la factorisation des descriptions de motifs d'architecture récurrents et la vérification des invariants d'architecture. Après avoir détaillé les principes de base de ces constructions, nous présentons le langage de requêtes d'architecture FPATH puis les extensions du langage FRACTAL ADL que nous avons définies.

6.3.1 Définition des invariants d'architecture

Notre proposition consiste à introduire deux nouvelles constructions dans le langage de description d'architectures FRACTAL ADL afin de faciliter la prise en compte des invariants d'architecture. Notre objectif est de définir un formalisme pour exprimer et isoler un motif d'architecture sous une forme canonique. Cette forme canonique peut ensuite être déclinée et appliquée sur une architecture finale. L'utilisation de cette forme canonique réifie différents invariants architecturaux définis par l'architecte de l'application. Chaque invariant est appliqué automatiquement sur l'architecture en filtrant les éléments de l'architecture avec une requête d'interrogation. La première construction du langage que nous définissons s'appuie sur les éléments filtrés par la requête d'interrogation pour intégrer une déclinaison de la forme canonique dans l'architecture. La seconde construction du langage que nous présentons est une assertion qui utilise le résultat de la requête d'interrogation pour déterminer si la condition exprimée sur l'architecture est respectée ou non et, le cas échéant, déclenche une erreur de chargement de la définition.

De la même façon qu'un invariant représente une condition qui ne doit pas varier pour que le système fonctionne correctement, les invariants d'architecture sont pris en compte dans notre proposition à l'aide de deux nouvelles constructions du langage FRACTAL ADL. Celles-ci permettent de générer un motif architectural qui doit être appliqué systématiquement et de vérifier qu'une propriété architecturale est bien respectée lors du chargement de la définition d'une application et lors des reconfigurations dynamiques de l'architecture. La requête d'interrogation que nous utilisons permet de capturer les élements caractéristiques de la description FRACTAL ADL (par exemple, composants, interfaces, attributs). Chaque élément de l'architecture capturé par la requête alimente les constructions du langage que nous définissons.

6.3.2 FPATH : une syntaxe pour la navigation architecturale

FPATH est un sous-langage de FSCRIPT qui correspond aux expressions qui permettent de capturer des éléments de l'architecture de l'application, mais pas de la modifier [DL06]. Le langage FPATH a été défini dans la thèse de Pierre-Charles David [Dav05] et il permet de naviguer dans une architecture à base de composants FRACTAL en cours d'exécution. Notre proposition consiste à conserver la syntaxe de FPATH en l'appliquant à la navigation dans les descriptions des architectures à base de composants. De cette manière, FPATH fournit une syntaxe complète pour capturer les éléments caractéristiques d'une architecture et constitue un bon candidat pour réaliser nos requêtes architecturales.

FPATH se présente sous la forme d'une notation simple et expressive, inspirée de XPATH [Wor99], pour la navigation dans une architecture FRACTAL et la sélection d'éléments (composants, interfaces, attributs) répondant à certains critères. Le langage repose sur la modélisation d'un ensemble de composants FRACTAL sous la forme d'un graphe orienté, dont les nœuds représentent les composants, leurs interfaces et attributs, et dont les arcs sont annotés par des labels qui dénotent le type de relation entre deux nœuds (par exemple, interface, sous-composant, liaison). En plus des types d'expressions habituelles (arithmétiques, combinateurs booléens et comparaisons, etc.), FPATH ajoute des expressions de type chemins relatifs (à un composant de départ). Un chemin consiste en une suite de pas, chacun constitué de trois éléments : `axe::test[predicat]`. À chaque pas, un ensemble de nœuds de départ est converti en un nouvel ensemble en suivant les arcs du graphe identifié par l'`axe`, puis en filtrant le résultat grâce au `test` et aux `prédicats` optionnels.

Pour illustrer la syntaxe FPATH, nous utilisons l'exemple de l'application HelloWorld développée en utilisant le modèle de composants FRACTAL. L'architecture de cette application est rappelée dans la figure 6.2. Elle se compose de deux composants primitifs (composants Client et Server) connectés entre eux et d'un composant composite (composant HelloWorld) qui exporte l'interface serveur du composant Client. Le composant Serveur dispose d'un attribut header afin de configurer l'entête des messages qu'il affiche.

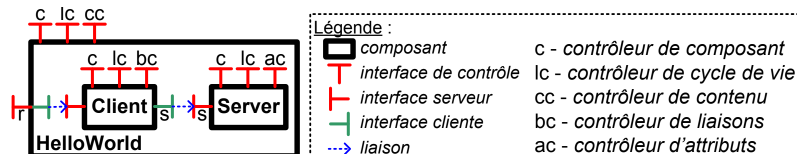


FIG. 6.2: Architecture de l'application HelloWorld.

En prenant pour exemple la figure 6.2, l'expression FPATH `Client/binding::s` sélectionne dans un premier temps le sous-composant (axe `child`) nommé `Client` du composant initial, puis suit la liaison (axe `binding`) dont le nom est `s` pour finalement retourner l'interface serveur qui lui est connectée. Le contexte initial utilisé pour la navigation est la définition courante (le composant HelloWorld dans cet exemple). De même, l'expression booléenne `count(internal-interface::*[required() and not(bound())]) > 0` renvoie vrai si et seulement si le composant initial possède des interfaces internes requises qui ne sont pas encore connectées.

La figure 6.3 fournit une représentation visuelle sous forme d'un graphe orienté des éléments caractéristiques d'une architecture utilisant le modèle de composants FRACTAL (par exemple, composants, interfaces, attributs) et des axes de base qui peuvent être utilisés pour naviguer entre ces éléments.

La syntaxe FPATH définit également un certain nombre d'axes de synthèse construits avec les axes de base et permettant de simplifier l'écriture de certaines expressions plus complexes. La liste des axes de base et des axes de synthèse est résumée dans le tableau 6.1.

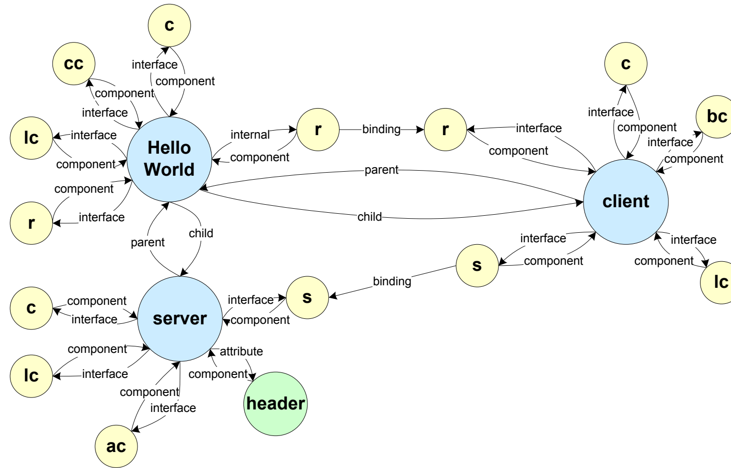


FIG. 6.3: Représentation graphique des chemins FPATH pour l'application HelloWorld.

Libellé	Sémantique
component	Relie un nœud interface ou attribut à celui du composant auquel appartient l'interface ou l'attribut.
interface	Relie un nœud composant à chacune de ses interfaces.
attribute	Relie un nœud composant à chacun de ses attributs de configuration.
binding	Relie deux interfaces si et seulement si les deux interfaces sont connectées. L'orientation de l'arc correspond à l'orientation de la connexion.
child	Relie deux composants A et B, dans cette direction, si et seulement si B est un sous-composant direct du composite A.
parent	Symétrique de child. Relie les composants A et B si et seulement si B est un des parents (super-composants) directs de A.
child-or-self	Similaire à child, en incluant le composant de départ.
parent-or-self	Similaire à parent, en incluant le composant de départ.
descendant	Sélectionne tous les sous-composants directs et indirects du composant de départ. Correspond à la clôture transitive de l'axe child.
descendant-or-self	Similaire à descendant, en incluant le composant de départ.
ancestor	Sélectionne tous les parents directs et indirects du composant de départ. Correspond à la clôture transitive de l'axe parent.
ancestor-or-self	Similaire à ancestor, en incluant le composant de départ.
sibling	Sélectionne tous les composants qui se trouvent « au même niveau » que le composant de départ, c.-à-d. qui sont sous-composants directs d'au moins un des parents directs du composant de départ. Il s'agit en fait d'un raccourci syntaxique pour `parent::*/*[.!=$c]` (où `$c` représente le composant de départ).
sibling-or-self	Similaire à sibling, mais inclut aussi le composant de départ.
external-interface	Variante de l'axe interface qui ne sélectionne que les interfaces externes.
internal-interface	Variante de l'axe interface qui ne sélectionne que les interfaces internes.

TAB. 6.1: Description des axes FPATH.

6.3.3 Foreach : génération de motifs architecturaux pour FRACTAL ADL

En combinant les propriétés d'extensibilité de FRACTAL ADL (présenté dans la section 3.3.1 du chapitre 3) et le pouvoir d'expression de la syntaxe FPATH, nous proposons de définir l'opérateur `foreach` comme un nouvel élément de la syntaxe FRACTAL ADL. Cet opérateur s'intègre dans la grammaire actuelle de FRACTAL ADL et permet de reproduire un motif de construction ADL dans différents contextes. Cet opérateur agit sur un ensemble de noeuds de l'AST FRACTAL ADL[1] filtrés à l'aide de la syntaxe FPATH et applique sur la définition courante un morceau de définition ADL paramétré par le résultat de la requête FPATH. La syntaxe de cet opérateur est la suivante :

```
<foreach variable="var" values="expression">
<!--[...description Fractal ADL...] -->
</foreach>
```

où `var` représente le nom de la variable dans laquelle un des nœuds capturés par la requête FPATH `expression` est stocké pour chaque itération. La valeur de cette variable peut être récupérée depuis les éléments de définition contenus entre les balises `<foreach>` en utilisant la syntaxe `%{$var}`.

Le listing 6.2 présente la syntaxe complète de l'opérateur `foreach`. Cette syntaxe se base sur le formalisme de définition de document DTD (en anglais, *Document Type Definition*) utilisé pour définir la grammaire de FRACTAL ADL. Un élément `foreach` accepte les mêmes éléments que l'élément `definition` de FRACTAL ADL (par exemple, `comment`, `component`, `binding`). L'élément `foreach` est paramétré par deux attributs `variable` et `values`. L'attribut `variable` définit le nom de la variable dans laquelle les éléments filtrés par la requête FPATH associée à l'attribut `values` seront stockés.

```
1 <!ELEMENT foreach (
2   comment*, interface*, component*, binding*,
3   content?, attributes?, controller?, template-controller?,
4   logger?, virtual-node?
5 ) >

7 <!ATTLIST foreach
8   variable CDATA #REQUIRED
9   values   CDATA #REQUIRED
10 >
```

LST. 6.2: Description de la syntaxe de l'opérateur `foreach`.

La sémantique d'application de cet opérateur spécifie que lorsqu'il est utilisé dans une hiérarchie de composants, les opérateurs `foreach` des composants fils de la hiérarchie doivent être traités avant ceux des composants parents. Cette sémantique permet de décrire des motifs d'architecture au niveau des composants parents reposant sur des motifs générés au niveau des composants fils (par exemple, la génération d'un motif d'exportation automatique des interfaces serveur d'un composant fils générées à partir d'un autre motif).

Ainsi le listing 6.3 présente un exemple de réécriture d'un sous-ensemble de la définition FRACTAL ADL `MySequence` en utilisant l'opérateur `foreach` pour remplacer la déclaration statique des liaisons par une liaison contextuelle, c.-à-d. liaison dépendante des composants déclarés dans l'architecture. Cette nouvelle définition est nommée `MyForeachSequence` dans le listing 6.3. Ainsi, le motif `foreach` employé dans cet exemple (lignes 20–23) permet de connecter automatiquement toutes les étapes au composant `runner`.

[1] L'AST FRACTAL ADL est présenté dans la section 3.3.1 du chapitre 3

```
1 <definition name="MySequence">
2   <!-- [...] -->
3   <component name="runner" definition="SequenceStrategy"/>
4   <component name="step1" definition="Step"/>
5   <component name="step2" definition="Step"/>
6   <component name="step3" definition="Step"/>
7   <!-- [...] -->
8   <binding client="runner.runnable-step1" server="step1.run"/>
9   <binding client="runner.runnable-step2" server="step2.run"/>
10  <binding client="runner.runnable-step3" server="step3.run"/>
11 </definition>

13 <definition name="MyForeachSequence">
14  <!-- [...] -->
15  <component name="runner" definition="SequenceStrategy"/>
16  <component name="step1" definition="Step"/>
17  <component name="step2" definition="Step"/>
18  <component name="step3" definition="Step"/>
19  <!-- [...] -->
20  <foreach variable="step"
21           values="name(runner/sibling::*[server(interface::run)])">
22    <binding client="runner.runnable-%{$step}" server="%{$step}.run"/>
23  </foreach>
24 </definition>
```

LST. 6.3: Description FRACTAL ADL du composant MySequence.

6.3.4 Assert : vérification d'invariants architecturaux pour FRACTAL ADL

En combinant les propriétés d'extensibilité de FRACTAL ADL et le pouvoir d'expression de la syntaxe FPATH, nous proposons de définir l'opérateur `assert` comme un nouvel élément de la syntaxe FRACTAL ADL. Cet opérateur s'intègre dans la grammaire actuelle de FRACTAL ADL et permet de vérifier le respect d'un invariant d'architecture dans différents contextes. Ainsi, l'opérateur `assert` parcourt les nœuds de l'AST FRACTAL ADL en appliquant une expression booléenne FPATH. Si cette expression booléenne n'est pas vérifiée, alors l'opérateur déclenche une erreur de chargement dont le message est paramétrable. La syntaxe de cet opérateur est la suivante :

```
<assert condition="expression" message="msg"/>
```

où `condition` représente l'expression booléenne FPATH à vérifier et `msg` est le message d'erreur à afficher en cas de non respect de l'invariant exprimé par l'expression booléenne.

Le listing 6.4 présente la syntaxe complète de l'opérateur `assert`. Cette syntaxe se base sur le formalisme de définition de document DTD utilisé pour définir la grammaire de FRACTAL ADL. L'élément `assert` n'accepte pas d'élément fils et définit deux attributs `condition` et `message` représentant respectivement l'invariant d'architecture sous la forme d'une expression booléenne FPATH et le message d'erreur associé sous la forme d'une chaîne de caractères.

```
1 <!ELEMENT assert EMPTY >

3 <!ATTLIST assert
4   condition CDATA #REQUIRED
5   message   CDATA #REQUIRED
6 >
```

LST. 6.4: Description de la syntaxe de l'opérateur `assert`.

Contrairement à l'opérateur `foreach`, l'ordre d'application des opérateurs `assert` importe peu dans la mesure où l'opérateur `assert` n'a pas d'effet de bord sur l'architecture décrite. Par contre, tout opérateur `assert` doit être évalué une fois que tous les opérateurs `foreach` ont été exécutés. Cette sémantique permet de vérifier les effets de bord de l'application des opérateurs `foreach` pour garantir la validité des motifs générés (par exemple, la vérification que l'exportation automatique des interfaces serveur sur un composant a été réalisée pour au moins une

interface).

Le listing 6.5 présente un exemple de vérification d'un invariant d'architecture sur la définition FRACTAL ADL `MySequence`. Ainsi, la définition `MyAssertSequence` étend la définition `MySequence` pour introduire un invariant d'architecture précisant que l'instance de composant `SequenceStrategy` doit être connectée à au moins une instance de composant `Step`.

```
1 <definition name="MyAssertSequence" extends="MySequence">
2   <assert condition="count(runner/sibling::*[server(interface::run)]) > 0"
3           message="The Sequence strategy requires at least one component with a run interface"/>
4 </definition>
```

LST. 6.5: Description FRACTAL ADL du composant MySequence.

6.4 Illustration de l'utilisation des invariants architecturaux

Cette section illustre l'utilisation des opérateurs de génération de motifs et de vérification d'invariants d'architecture basés sur la syntaxe FPATH. Dans un premier temps, l'exemple du composant MySequence est adapté pour rendre sa définition plus modulaire grâce à l'utilisation de nos opérateurs. Ensuite, des définitions génériques correspondant à des motifs caractéristiques d'une architecture à base de composants sont présentées.

6.4.1 Application à la définition du composant MySequence

Dans cette section, nous illustrons les bénéfices de notre langage de description et de vérification de motifs d'architecture sur l'exemple du composant MySequence introduit dans la section 6.2. Nous proposons ainsi de diviser la définition du composant MySequence en plusieurs descriptions indépendantes identifiant chacune une préoccupation particulière.

Le listing 6.6 adresse la préoccupation de la gestion de la trace et de son intégration. Ainsi la définition `AutoLogger` se compose d'un composant `logger` chargé de collecter les traces et d'un motif d'architecture décrivant l'intégration du composant `logger`. Ce motif recherche les noms des composants disposant d'une interface cliente nommée `logger` en utilisant la requête FPATH `*[client(interface::logger)]`. Pour chaque composant filtré par la requête, une liaison est créée entre le composant retourné par la requête FPATH et le composant `logger`. Notons que la requête FPATH présentée ici est minimaliste, celle-ci peut être étendue pour vérifier que le type de l'interface filtrée est compatible avec l'interface serveur `log` du composant `logger` et que celle-ci n'est pas déjà connectée. L'assertion définie dans cette description assure qu'au moins un autre composant est connecté au composant `logger`.

```
1 <definition name="AutoLogger">
2   <component name="logger" definition="Logger"/>
3   <foreach variable="component" values="name(*[client(interface::logger)])">
4     <binding client="%{$component}.logger" server="logger.log"/>
5   </foreach>
6   <assert condition="count(*[client(interface::logger)]) > 0"
7           message="AutoLogger requires at least one component with a 'logger' interface"/>
8 </definition>
```

LST. 6.6: Description FRACTAL ADL de l'intégration du composant Logger.

La définition `Sequence` présentée dans le listing 6.7 décrit l'intégration d'un mécanisme de gestion d'une séquence d'étapes dans une architecture à base de composants. Ces étapes correspondent à n'importe quel type de composant fournissant une interface serveur *run* de

type `java.lang.Runnable`. Pour ce faire, la définition `Sequence` définit une interface serveur `run` qu'elle connecte au composant `runner`. Le composant `runner` correspond à la stratégie d'exécution de la séquence d'étapes. L'élément `foreach` est utilisé dans cette description pour décrire les composants qui peuvent être connectés à l'interface `runnable` du composant `runner`. La requête FPATH `runner::sibling::*[server(interface::run)]` filtre les noms de composants différents du composant `runner` qui fournissent une interface `run`. Une fois encore la requête peut être complétée pour que le filtre ne s'exécute pas sur le nom de l'interface mais sur son type. L'assertion que nous définissons garantit que la séquence est connectée à au moins une étape.

```
1 <definition name="Sequence">
2   <interface name="run" role="server" signature="Runnable"/>
3   <component name="runner" definition="SequenceRunner"/>
4   <binding client="this.run" server="runner.run"/>
5   <foreach variable="step"
6            values="name(runner/sibling::*[server(interface::run)])">
7     <binding client="runner.runnable-%{$step}" server="%{$step}.run"/>
8   </foreach>
9   <assert condition="count(runner/sibling::*[server(interface::run)]) > 0"
10          message="Sequence requires at least a component with a 'run' server interface"/>
11 </definition>
```

LST. 6.7: Description FRACTAL ADL générique d'une séquence de composants.

Le listing 6.8 présente l'intégration du système d'information dans une architecture à base de composants. Cette intégration se compose de la définition du composant `SI` correspondant au système d'information et d'une règle de partage des instances. Cette règle spécifie que pour tout composant de l'architecture contenant un composant fils `si`, l'instance contenue dans ce composant sera la même que l'instance du composant `SI` intégrée par la définition `MySI`. Dans la mesure où l'intégration du composant `SI` vise à unifier les instances du système d'information manipulé par les différentes étapes, l'assertion que nous définissons vérifie qu'il existe au moins un composant dont le système d'information peut être partagé.

```
1 <definition name="MySI">
2   <component name="si" definition="SI"/>
3   <foreach variable="step" values="name(*[si])">
4     <component name="%{$step}">
5       <component name="si" definition="./si"/>
6     </component>
7   </foreach>
8   <assert condition="count(*[si]) > 0"
9           message="MySI requires at least one component using the SI definition"/>
10 </definition>
```

LST. 6.8: Description FRACTAL ADL de l'intégration du composant SI.

La définition `MySequence` décrite dans le listing 6.9 correspond à la définition finale de l'architecture. Cette définition intègre les motifs d'architecture `Sequence`, `MySI` et `AutoLogger` définis précédemment et les complète avec la définition des différentes étapes à réaliser dans la séquence. Ces différentes étapes sont les composants `step1`, `step2`, `step3` et `step4`.

```
1 <definition name="MySequence" extends="Sequence,MySI,AutoLogger">
2   <component name="step1" definition="Step1"/>
3   <component name="step2" definition="Step2"/>
4   <component name="step3" definition="Step3"/>
5   <component name="step4" definition="Step4"/>
6 </definition>
```

LST. 6.9: Intégration des motifs Sequence, SI et AutoLogger dans une séquence d'étapes.

La définition `MySequence` présentée dans le listing 6.9 est équivalente à la définition présenté dans le listing 6.1. La figure 6.4 décrit l'intégration des différentes préoccupations identifiées dans les définitions AutoLogger, MySI et Sequence avec la définition MySequence. Lors de l'application de la définition Sequence, tous les composants de la définition MySequence disposant d'une interface serveur *run* seront automatiquement connectés au composant Sequence Strategy. Lors de l'application de la définition MySI, les composants contenant un composant SI (dans cet exemple, les composants step1, step2 et step3) partageront cette instance avec l'instance locale du composant SI. Enfin, lors de l'application de la définition AutoLogger, les composants requiérant une interface *logger* seront automatiquement connectés au composant Logger. Cette intégration concerne les composants step1, step2 et step4 de la définition MySequence mais également le composant SI de la définition MySI.

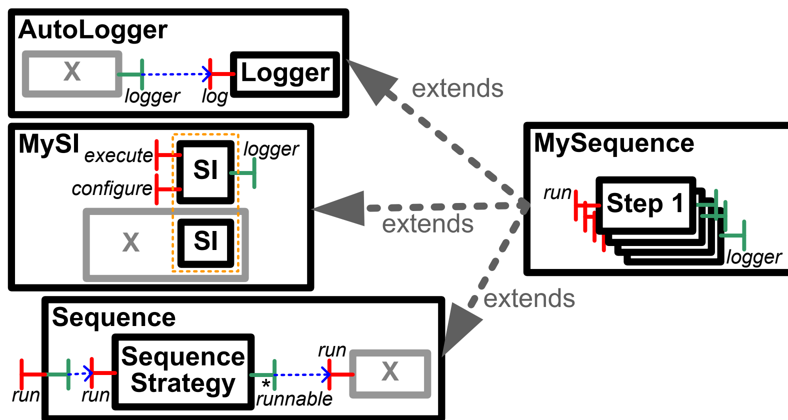


FIG. 6.4: Représentation modulaire du composant MySequence.

La définition d'une nouvelle étape dans la séquence consiste à développer le composant Step associé et à le déclarer dans le composant MySequence. Dès lors, la liaison avec les composants runner et logger ainsi que le partage du composant si seront automatiquement pris en charge par les motifs associés au composant MySequence. Par conséquent, notre approche simplifie (grâce à notre opérateur `foreach`) et fiabilise (grâce à notre opérateur `assert`) l'intégration de nouveaux composants dans une architecture existante.

La définition du composant MySequence en modules indépendants nous permet de réutiliser certains de ces modules dans des contextes différents. Ainsi, les définitions AutoLogger et Sequence peuvent être facilement généralisées pour être intégrées dans d'autres architectures. L'utilisation de ces différentes définitions nous permet d'isoler des préoccupations de différents niveaux (des préoccupations techniques à celles liées au métier de l'application). Certains de ces motifs d'architecture peuvent également être abstraits et paramétrés afin de factoriser leur définition et leur réutilisation.

6.4.2 Définition de motifs d'architecture génériques

Dans cette section, nous présentons différents motifs d'architecture génériques. Les paramètres de ces motifs sont exprimés en utilisant les arguments des définitions FRACTAL ADL. Ces motifs seront intensivement réutilisés dans les deux chapitres suivants.

Le listing 6.10 présente une définition pour l'automatisation de l'intégration des interfaces serveur de type `collection`. Les paramètres de la définition `AutoExport` sont le nom et le type de l'interface à intégrer dans l'architecture. La définition se charge ensuite d'ajouter l'interface dans l'architecture (ligne 2), de sélectionner (ligne 3) et de connecter les interfaces serveurs compatibles avec l'interface intégrée parmi les composants de l'architecture (lignes 4–5). L'utilisation d'un filtre sur le type de l'interface (et non pas le nom de l'interface) requiert l'utilisation des expressions FPATH dans les attributs de l'élément `binding` afin de naviguer dans

```
1 <definition name="AutoExport" arguments="name,type">
2   <interface name="${name}" role="server" cardinality="collection" signature="${type}"/>
3   <foreach variable="itf" values="*/interface::*[server() and compatible(${type})])">
4     <binding client="this.${name}-%{name($itf/parent::*)}"
5              server="%{name($itf/parent::*)}.%{name($itf)"/>
6   </foreach>
7   <assert condition="bound(internal-interface::*[startsWith(${name})])"
8           message="The interface ${name} is not exported."/>
9 </definition>

11 <definition name="Sequence" extends="AutoExport(run,java.lang.Runnable)">
12   <component name="runner" definition="SequenceRunner"/>
13   <!-- [...] -->
14 </definition>
```

LST. 6.10: Motif FRACTAL ADL de l'exportation d'une interface serveur collection.

les nœuds interfaces filtrés pour obtenir le nom de l'interface et le nom du composant associé. L'assertion utilisée ici (lignes 7–8) vérifie qu'au moins une interface du composant a été exportée via l'interface collection. Le listing 6.10 présente ensuite une application de la définition `AutoExport` (lignes 11–14) permettant d'exporter automatiquement toutes les interfaces serveur de type `java.lang.Runnable` de composants contenus dans le composant `Sequence` sous la forme d'une interface serveur collection nommée `run`.

Exportation automatique des interfaces serveurs d'un composant

Le listing 6.11 présente une définition pour l'automatisation de l'exportation des interfaces serveurs d'un composant. Les paramètres de la définition `AutoInclude` sont le nom (`component`) et la définition (`definition`) du composant à intégrer dans l'architecture et dont les interfaces serveurs doivent être exportées. La définition se charge ensuite d'ajouter le composant `component` dans l'architecture (ligne 2), de sélectionner (ligne 3), de définir (ligne 4) et d'exporter toutes les interfaces serveurs du composant `component` (ligne 5). L'assertion utilisée ici (lignes 7–8) vérifie qu'au moins une interface du composant `component` a été exportée. Le listing 6.11 présente ensuite une application de la définition `AutoInclude` (lignes 11–13) permettant d'exporter automatiquement toutes les interfaces serveurs du composant `runner` décrites dans la description FRACTAL ADL `SequenceRunner`.

```
1 <definition name="AutoInclude" arguments="component,definition">
2   <component name="${component}" definition="${definition}"/>
3   <foreach variable="itf" values="${component}/interface::*[server()]">
4     <interface name="%{name($itf)}" role="server" signature="%{signature($itf)}"/>
5     <binding client="this.%{name($itf)}" server="${component}.%{name($itf)}"/>
6   </foreach>
7   <assert condition="count(${component}/interface::*[server()]) > 0"
8           message="The component ${component} is not exported."/>
9 </definition>

11 <definition name="Sequence" extends="AutoInclude(runner,SequenceRunner)">
12   <!-- [...] -->
13 </definition>
```

LST. 6.11: Motif FRACTAL ADL de l'exportation des interfaces serveurs d'un composant.

Importation automatique d'une interface requise

Le listing 6.12 présente l'opération inverse consistant à importer automatiquement une interface cliente de l'architecture. Il s'agit ici d'une interface cliente de type singleton dont la référence peut être requise par plusieurs composants de l'architecture. La

définition AutoImport déclare l'interface cliente requise (ligne 2), sélectionne dans l'architecture les interfaces clientes compatibles (lignes 3–4) et connecte celles-ci à l'interface intégrée (ligne 5). La requête FPATH utilisée pour sélectionner les interfaces clientes compatibles (*/interface::*[client() and not(bound()) and compatible(${type})]) vérifie que les interfaces clientes filtrées ne sont pas déjà connectées (afin de ne pas remplacer une liaison définie par l'architecte). L'assertion associée à la définition AutoImport vérifie que les interfaces clientes dont le type est spécifié dans l'argument type sont toutes connectées (lignes 7–8). Le listing 6.12 propose une intégration différente de la fonction de trace (ligne 11). Cette intégration se différencie par le fait qu'aucun composant de trace n'est déclaré dans le composant composite, mais les interfaces de type Logger sont plutôt exportées via une interface cliente singleton déclarée sur le composant composite.

```
1 <definition name="AutoImport" arguments="name,type">
2   <interface name="${name}" role="client" signature="${type}"/>
3   <foreach variable="itf"
4    values="*/interface::*[client() and not(bound()) and compatible(${type})]">
5     <binding client="%{name($itf/parent::*)}.%{name($itf)}" server="this.${name}"/>
6   </foreach>
7   <assert condition="*/interface::*[client() and bound() and compatible(${type})]"
8    message="The interface ${name} is not imported."/>
9 </definition>

11 <definition name="AutoLogger" extends="AutoImport(logger,Logger)"/>
```

LST. 6.12: Motif FRACTAL ADL de l'importation d'une interface cliente.

Partage automatique des composants communs

La définition AutoSharing présentée dans le listing 6.13 permet de déclarer une instance de composant partageable (lignes 1–8). Ce type de définition permet de déclarer une instance de composant et de partager automatiquement celle-ci avec les sous-composants compatibles de l'architecture. La requête d'intégration utilisée dans cette définition filtre les sous-composants sur un niveau de hiérarchie dont le nom est identique au composant intégré. Le listing 6.13 revisite la définition MySI en utilisant la définition générique AutoSharing (lignes 10–13).

```
1 <definition name="AutoSharing" arguments="name,definition">
2   <component name="${name}" definition="${definition}"/>
3   <foreach variable="comp" values="name(*[${name}])">
4     <component name="%{comp}">
5       <component name="${name}" definition="./${name}"/>
6     </component>
7   </foreach>
8 </definition>

10 <definition name="MySI" extends="AutoSharing(si,SI)">
11   <assert condition="count(*/si) > 0"
12          message="MySI requires at least a component using the SI definition"/>
13 </definition>
```

LST. 6.13: Motif FRACTAL ADL du partage de composants.

Liaison automatique d'une collection d'interfaces requises

La définition AutoBind présentée dans le listing 6.14 réalise la liaison automatique d'une interface cliente collection d'un composant de l'architecture dont le nom, l'interface et le type sont spécifiés via les arguments de la définition AutoBind (lignes 1–9). Le composant de l'architecture spécifié par l'argument component est exclu de la recherche des interfaces serveurs compatibles (requête ${component}/sibling::*) pour éviter de créer une liaison bouclant sur le

même composant. Le listing 6.14 présente un exemple d'application de la définition générique `AutoBind` sur la définition `Sequence` (lignes 11–15). Cette définition simplifie non seulement la définition `Sequence` mais elle propose de réaliser la sélection des interfaces serveurs en fonction de leur signature au lieu d'utiliser le nom de l'interface.

```
1 <definition name="AutoBind" arguments="component,client,type">
2   <foreach variable="itf"
3    values="${component}/sibling::*/interface::*[server() and compatible(${type})]">
4     <binding client="${component}.${client}-%{name($itf/parent::*)}"
5              server="%{name($itf/parent::*}.%{name($itf)}"/>
6   </foreach>
7   <assert condition="${component}/interface::${client}[bound()]"
8    message="The interface ${client} is not bound."/>
9 </definition>

11 <definition name="Sequence" extends="AutoBind(runner,runnable,java.lang.Runnable)">
12   <interface name="run" role="server" signature="Runnable"/>
13   <component name="runner" definition="SequenceRunner"/>
14   <binding client="this.run" server="runner.run"/>
15 </definition>
```

LST. 6.14: Motif FRACTAL ADL de la liaison d'une interface cliente collection.

6.5 Éléments d'implantation

Cette section s'attache à la présentation des éléments d'implantation liés à la réalisation du langage de description de motifs d'assemblage. Cette réalisation se compose du langage d'interrogation d'architecture appelé FPATH, d'une extension de l'usine FRACTAL ADL, d'une extension du contrôle des composants FRACTAL et d'une extension des générateurs FRACLET.

6.5.1 Implantation de l'interpréteur FPATH

Le langage FPATH dans sa version initiale ne permet de réaliser des requêtes d'interrogation que sur une structure en cours d'exécution.

Notre proposition consiste à adapter ce langage afin d'exécuter les requêtes d'interrogation non seulement sur les définitions d'une architecture décrite avec le langage FRACTAL ADL mais aussi sur les composants FRACTAL en cours d'exécution. L'implantation actuelle de FPATH est intégrée dans l'outil FSCRIPT [DL06]. Par conséquent, nous proposons de fournir une implantation de FPATH indépendante de FSCRIPT et réalisée avec le modèle de composants FRACTAL. L'utilisation du modèle de composants FRACTAL dans l'implantation de FPATH nous permet de réutiliser les composants qui ne sont pas spécifiques au type d'interrogation que nous réalisons (architecture décrite en FRACTAL ADL ou architecture FRACTAL à l'exécution). De plus, l'utilisation des composants nous permet d'étendre ou de restreindre les capacités de l'outil en fonction des besoins de l'utilisateur (par exemple, configuration des axes de navigation, des filtres et des fonctions).

La figure 6.5 présente l'architecture globale de FPATH. Le composant FPATH est composé d'un analyseur syntaxique (composant Parser), d'une usine de nœuds (composant Node Factory) et d'un interpréteur de requêtes représenté sous la forme d'un répartisseur (composants Dispatcher, Path, Functions, Operators et Literals). Le composant Parser est responsable de l'analyse de la requête d'interrogation et de la création de la représentation mémoire de cette requête. La représentation mémoire que nous utilisons est un arbre de syntaxe abstraite (AST) dont les nœuds représentent des éléments caractéristiques de la requête (chemins FPATH, opérateurs de combinaison, appels de fonctions, valeurs primitives). L'analyseur syntaxique et l'AST associé peuvent être construits en utilisant des outils tels que JavaCC, SableCC ou ANTLR. Les classes générées par ces outils sont ensuite rendues compatibles avec le modèle de composants FRACTAL en intégrant les annotations FRACLET dans le processus de génération de l'analyseur syntaxique.

Le composant Node Factory est responsable de la création des différents nœuds représentant les résultats possibles des requêtes FPATH. L'implantation de ce composant dépend du type d'interrogation réalisée sur l'architecture. Par conséquent, il existe deux implantations de ce composant : l'une pour les nœuds représentant des artefacts à l'exécution et l'autre pour les nœuds représentant des artefacts décrits du modèle de composants FRACTAL. L'interpréteur de requête FPATH est divisé en quatre composants qui représentent les quatre éléments caractéristiques d'une requête FPATH. Le composant Dispatcher interprète une requête FPATH représentée sous la forme d'un AST (obtenu à partir du composant Parser). En fonction du type du nœud courant de l'AST, le composant Dispatcher oriente le traitement de la requête vers un opérateur (composant Operators), un appel de fonction (composant Functions), la création d'un litéral (composant Literals) ou vers l'exécution d'un chemin FPATH (composant Path). Le composant NodeFactory est partagé entre les composants de traitement de la requête afin d'unifier la gestion des types de nœuds FPATH manipulés par les requêtes. Enfin, le composant Path exporte une interface *variable* afin de pouvoir supporter la notion de chemin de navigation relatif.

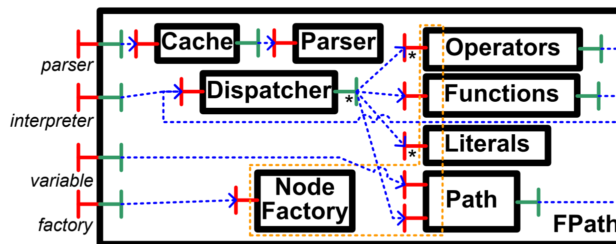


FIG. 6.5: Architecture de FPATH.

La figure 6.6 présente le composant en charge du traitement d'un chemin FPATH. Le composant Interpreter analyse le chemin FPATH. Pour chaque étape de ce chemin, le composant Interpreter délègue l'exploration de l'axe spécifié dans la requête au composant Axis. Ensuite, le composant Interpreter filtre l'ensemble de nœuds obtenus en utilisant le composant Tests. Enfin, si l'étape utilise un ou plusieurs prédicats, le composant Interpreter utilise l'interface importée *handler* pour filtrer l'ensemble de nœuds restants. Le composant Variables peut être utilisé par le composant Interpreter lorsque le chemin FPATH à analyser est un chemin relatif.

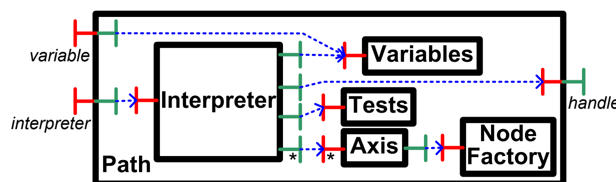


FIG. 6.6: Architecture des chemins FPATH.

La figure 6.7 détaille la conception des axes d'exploration de FPATH. Les axes de base sont regroupés dans le composant Backend. Ces axes de base sont dépendants du type de navigation à réaliser (navigation dans FRACTAL ADL ou navigation à l'exécution). Les axes de synthèse sont obtenus par composition des axes de base avec des opérateurs d'union ou de récursion (par exemple, l'axe *descendant-or-self* est obtenu par composition de l'axe child avec l'opérateur Loop pour obtenir l'axe de synthèse *descendant* puis par composition de l'opérateur Or avec l'axe Self).

Le listing 6.15 présente la définition du composant `Backend` regroupant les axes de base de FPATH. La définition présentée ici utilise les axes de base pour la navigation dans une architecture FRACTAL ADL. L'utilisation des définitions `AutoExport` et `AutoImport` permettent respectivement d'exporter les interfaces *axe* de chaque composant et d'importer la référence du composant `NodeFactory`. Cette structure extensible supporte l'intégration de nouveaux axes de

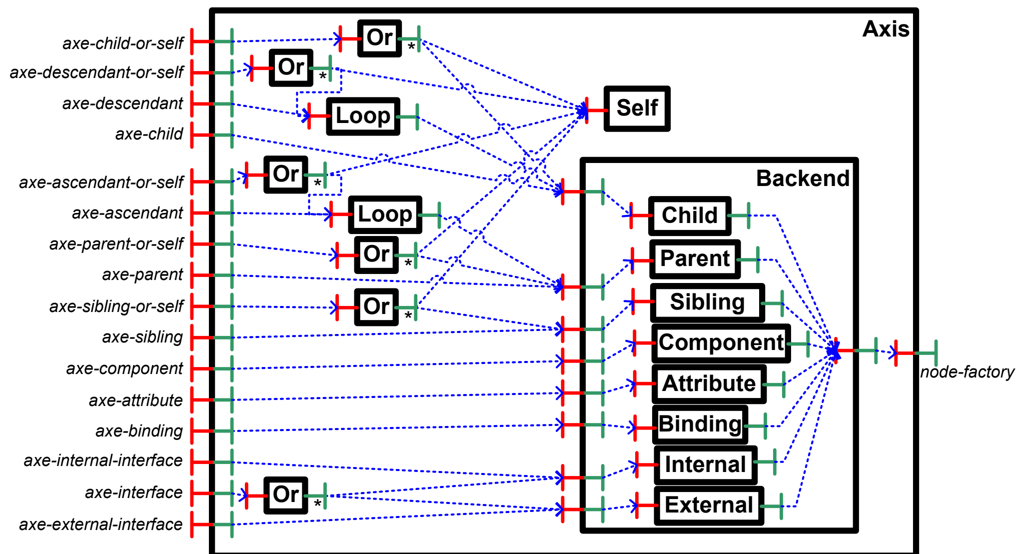


FIG. 6.7: Architecture des axes FPATH.

base. Par exemple, il est possible d'ajouter le support des interfaces à FPATH en intégrant un axe Method permettant d'accéder à la liste des méthodes disponibles sur une interface.

```
1 <definition name="Backend"
2 extends="AutoExport(axe,Axe),AutoImport(factory,NodeFactory)">
3   <component name="child"              definition="FractalAdlChild"/>
4   <component name="parent"             definition="FractalAdlParent"/>
5   <!-- [...] -->
6   <component name="internal-interface" definition="FractalAdlInternalInterface"/>
7   <component name="external-interface" definition="FractalAdlExternalInterface"/>
8 </definition>
```

LST. 6.15: Description FRACTAL ADL du composant Backend.

Une étape de chemin FPATH comporte non seulement un axe d'exploration mais aussi deux types de filtres. Le premier type de filtre est présenté dans la figure 6.8 et consiste à effectuer un filtrage sur le nom des éléments obtenus à l'issue de l'exploration de l'axe. Le composant Tests supporte quatre filtres identifiés par les composants Not, WildCard, RegExp et Name. La valeur du filtre correspond à l'élément `test` identifié dans les chemins FPATH. Le composant Not ne conserve que les nœuds dont le nom diffère de la valeur du filtre. Le composant WildCard conserve tous les nœuds si la valeur du filtre est égale à $\star$. Le composant Regexp extrait une expression régulière à partir de la valeur filtre et ne conserve que les nœuds dont le nom est compatible avec cette expression régulière. Enfin, le composant Name ne conserve que les nœuds dont le nom est identique à la valeur du filtre.

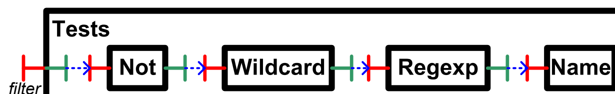


FIG. 6.8: Architecture des tests FPATH.

Le second type de filtre disponible dans la syntaxe FPATH correspond aux prédicats. Les prédicats correspondent à des expressions FPATH booléennes dont le résultat conditionne le filtrage des nœuds restants. Par conséquent, l'évaluation des prédicats est réalisée en déléguant

l'exécution de la requête FPATH contenue dans le prédicat à l'interface *handler* du composant Path.

La figure 6.9 présente la gestion des appels de fonctions dans les requêtes FPATH. Ces appels de fonctions sont généralement réalisés sur des nœuds FPATH. Le composant Functions dispose d'une interface *handler* afin d'analyser la valeur des paramètres d'un appel de fonction. Le composant Function Dispatcher détermine le type du nœud courant X associé à l'appel de fonction et délègue l'exécution de l'appel de la fonction au composant X functions associé. Si l'exécution de la fonction n'est pas réalisable, le composant Function Dispatcher utilise un mécanisme de réflexivité pour appeler la fonction sur l'objet associé au nœud FPATH.

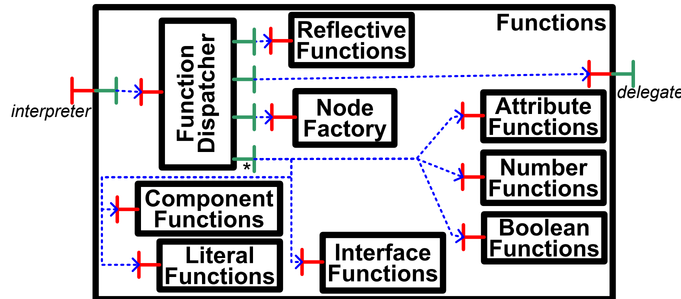


FIG. 6.9: Architecture des fonctions FPATH.

Enfin, la figure 6.10 présente le composant Operators responsable de la gestion des opérateurs de combinaison disponibles dans FPATH. Le composant Boolean Operators regroupe les opérateurs booléens, le composant Comparative Operators se charge des opérateurs de comparaison et le composant Arithmetic Operators supporte la définition des opérations arithmétiques. Ces opérations sont directement disponibles dans le composant FPATH par propagation des interfaces *interpreter*. Ainsi, le composant Dispatcher peut orienter le traitement de la requête FPATH sur le bon opérateur.

Discussion. Cette implantation de FPATH constitue une autre illustration de l'application des composants de fine granularité à la construction des intergiciels. Les composants de FPATH sont ainsi développés en utilisant notre modèle de programmation FRACLET et l'assemblage de ces composants exploite les constructions définies dans ce chapitre. Comparée à l'implantation existante de FPATH (développée en langage objet), notre implantation offre beaucoup plus de flexibilité. En effet, nous pouvons choisir et étendre les axes supportés dans les requêtes FPATH. Les types de filtres applicables dans les requêtes FPATH peuvent également être configurés. La liste des opérateurs de composition et des fonctions exécutables peut également être adaptée selon les besoins de l'utilisateur de l'outil FPATH. Enfin, en sélectionnant les composants Backend et Node Factory appropriés, il est possible de naviguer aussi bien sur des applications FRACTAL en cours d'exécution que sur des descriptions d'architectures basées sur le langage FRACTAL ADL, comme illustré dans la figure 6.11.

6.5.2 Extension de l'usine FRACTAL ADL

L'intégration des opérateurs `foreach` et `assert` dans FRACTAL ADL nécessite non seulement de modifier la description de la grammaire mais aussi le composant `Loader` de FRACTAL ADL afin d'intégrer deux nouveaux composants Foreach Loader et Assert Checker dans la chaîne de délégation de l'usine FRACTAL ADL. Comme illustré dans la figure 6.12, le composant Foreach Loader est placé entre les composants Binding Loader et Component Loader afin de générer les motifs d'architecture une fois que les composants ont été chargés par le composant Loader mais avant de vérifier la validité de l'architecture. Celui-ci transforme un nœud de l'AST de type `foreach` en un ensemble de clones des nœuds associés aux éléments fils de l'opérateur

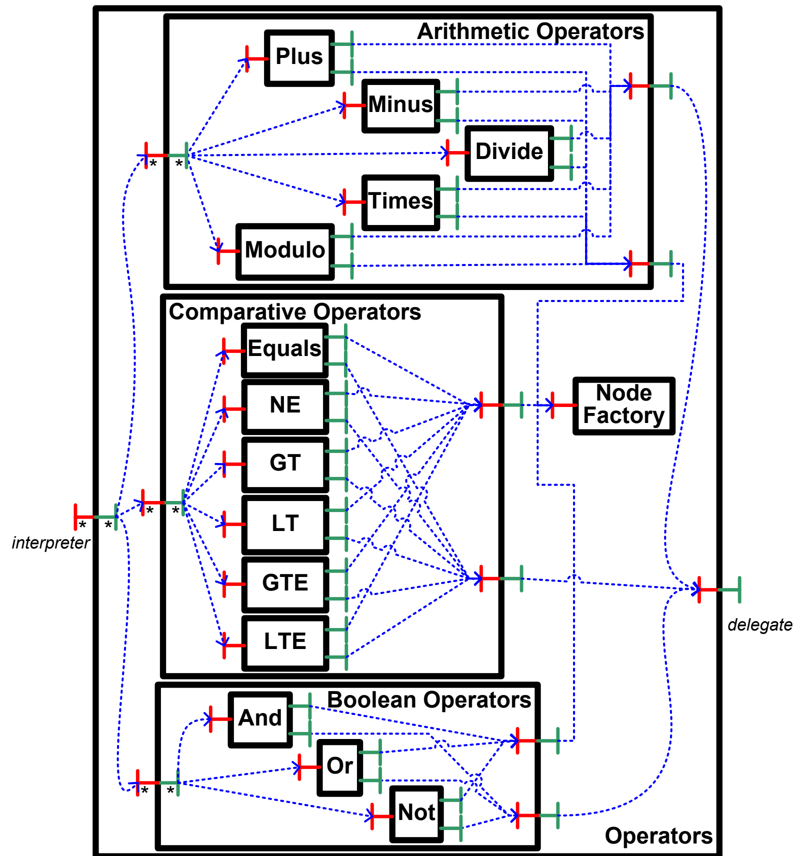


FIG. 6.10: Architecture des opérateurs FPATH.

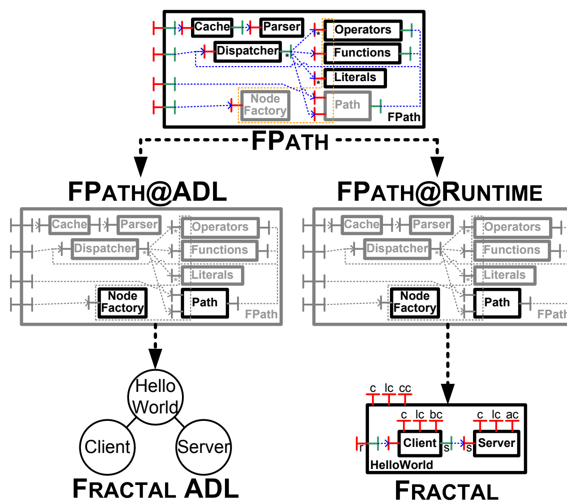


FIG. 6.11: Configurations de l'interpréteur FPATH.

foreach dont les variables sont remplacées par le résultat de la requête FPATH. Le composant Assert Checker est placé en tête de la chaîne de délégation afin de vérifier les invariants d'architecture une fois que la description architecturale est finalisée et que tous les opérateurs foreach ont été appliqués. Ce composant vérifie pour chaque nœud de l'AST de type assert que la condition associée est exacte.

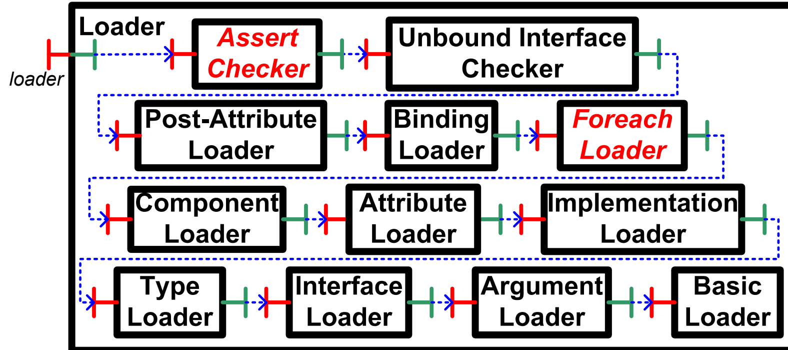


FIG. 6.12: Extension du composant Loader FRACTAL ADL.

La figure 6.13 présente l'architecture des composants Foreach Loader et Assert Checker. Afin de réduire l'empreinte mémoire liée à l'intégration de notre proposition, il est possible de partager l'instance du composant FPATH requis par chacun des opérateurs. Chaque composant récupère l'AST FRACTAL ADL chargé après invocation de l'interface *client-loader*, effectue les traitements qui lui sont associés sur l'AST, puis retourne celui-ci au composant situé en amont de la chaîne de délégation.

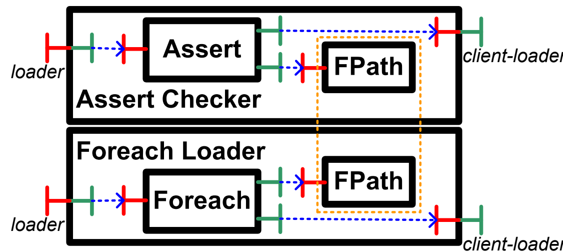


FIG. 6.13: Architecture des composants Foreach Loader et Assert Checker FRACTAL ADL.

6.5.3 Extension du contrôle des composants FRACTAL

L'objectif de notre extension du contrôle des composants FRACTAL vise à conserver les invariants définis dans les descriptions FRACTAL ADL durant l'exécution du composant. La connaissance de ces invariants permet de vérifier que toute reconfiguration *ad-hoc* du composant respecte les invariants d'architecture définis par l'architecte.

Le support des invariants à l'exécution nécessite d'étendre le modèle de composants FRACTAL avec un nouveau contrôleur dédié au contrôle des invariants. La réalisation du contrôleur est basée sur l'implantation de deux interfaces, illustrées dans le listing 6.16. L'interface de contrôle InvariantController définit les méthodes permettant de définir (avec la méthode addFcInvariant()), supprimer (avec la méthode removeFcInvariant()) et lister (avec la méthode listFcInvariant()) les invariants associés à un composant FRACTAL. L'interface InvariantController est accessible sur le composant tandis que l'interface InvariantCoordinator correspond à une interface interne à la membrane du composant.

En effet, la méthode `checkFcInvariant()` permet de vérifier que les invariants associés à un composant ne sont pas violés par la configuration actuelle.

```
1 public interface InvariantController {
2   void addFcInvariant(String invariant, String message);
3   void removeFcInvariant(String invariant) throws InvariantException;
4   String[] listFcInvariant();
5 }
6 public interface InvariantCoordinator {
7   void checkFcInvariant() throws InvariantException;
8 }
```

LST. 6.16: Description des interfaces de contrôle des invariants.

L'architecture et le comportement du contrôleur d'invariants est présenté dans la figure 6.14. Lors de l'ajout d'un nouvel invariant, le contrôleur compile l'invariant sous la forme d'une requête FPATH en utilisant l'interface parser du composant FPath. Lorsque le composant est démarré (via la méthode `startFc()` du contrôleur `LifeCycleController`), la méthode `checkFcInvariant()` de l'interface `InvariantCoordinator` est invoquée pour vérifier que la configuration est valide. La méthode `checkFcInvariant()` utilise alors la configuration du composant FPath permettant de naviguer dans une architecture à l'exécution. Si l'un des invariants n'est pas respecté par la configuration actuelle, une exception `InvariantException` est levée puis convertie par le contrôleur de cycle de vie en une exception `LifeCycleException`.

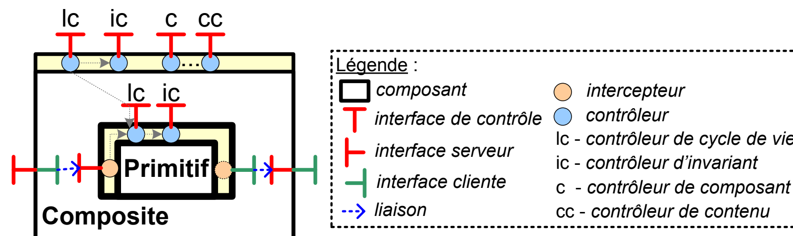


FIG. 6.14: Intégration du contrôleur d'invariants à la membrane des composants FRACTAL.

L'implantation de ce contrôleur peut être réalisée avec les implantations JULIA [BCL+06] ou AOKELL [SPDC06] du modèle de composants FRACTAL. La réalisation du contrôleur avec JULIA consiste à développer un ensemble de *mixin* pour développer le contrôleur d'invariants et l'extension du contrôleur de cycle de vie. La définition de la nouvelle membrane des composants est réalisée en modifiant le fichier de configuration de JULIA. La réalisation du contrôleur avec AOKELL permet de structurer l'architecture de la membrane et de faire apparaître le composant FPath dans cette architecture. La définition de la membrane des composants est réalisée dans AOKELL par un assemblage de composants techniques en utilisant FRACTAL ADL. L'utilisation de ce composant FPath par tous les contrôleurs d'invariants des composants, propose un cas d'utilisation du partage de composant entre les composants composites modélisant les membranes.

L'intégration du support du contrôleur d'invariants dans l'usine FRACTAL ADL nécessite d'étendre le contenu du composant Builder afin d'ajouter la conversion des nœuds `assert` de l'AST FRACTAL ADL associés à un composant en une séquence d'appels à la méthode `addFcInvariant()` si l'interface `InvariantController` est fournie par la membrane du composant.

6.5.4 Extension des générateurs FRACLET

Les invariants d'architectures peuvent également être réalisés au niveau des composants primitifs d'une application. Ce type de définition peut permettre de vérifier qu'une interface cliente

collection doit être liée à exactement *n* composants ou que la valeur d'un attribut de composant ne peut excéder une borne limite. Pour ce faire, nous définissons une nouvelle annotation @assert pour marquer les composants d'une application. Cette annotation définit deux attributs *condition* et *message* spécifiant respectivement l'invariant d'architecture et le message d'erreur à afficher en cas de non-respect de l'invariant. Cette annotation @assert est interprétée par le générateur de définitions primitives FRACTAL ADL pour intégrer automatiquement une balise `<assert/>` dans la définition du composant pour chaque invariant déclaré dans le code du composant.

Finalement, FRACLET a également été étendu pour intégrer un nouveau générateur de descriptions FRACTAL ADL. Ce générateur étend le générateur de descriptions composite abstrait pour prendre en charge la liaison automatique des composants compatibles avec l'interface cliente collection d'un composant. Dès lors, si un composant `Client` requiert une interface collection, une description FRACTAL ADL `ClientAutoBind` est générée automatiquement par FRACLET. Cette description intègre un opérateur `foreach` qui sélectionne tous les composants co-localisés avec le composant primitif et qui fournissent une interface serveur compatible avec le type de l'interface cliente collection. Cette opération est réalisée en utilisant la définition générique `AutoBind` définie dans le listing 6.14.

Le listing 6.17 présente un exemple de description ADL obtenue avec ce nouveau générateur. Ce générateur connecte automatiquement l'interface cliente `server` de type collection requise par le composant `client` avec les interface de type `Service` fournies par les composants co-localisés.

```
1 <definition name="ClientAutoBind" extends="ClientComposite,AutoBind(client,server,Service)"/>
```

LST. 6.17: Description FRACTAL ADL de la description générée ClientAutoBind.

6.5.5 Discussion

À l'heure actuelle, notre solution n'est pas complétement implantée mais un prototype d'implantation du langage FPATH avec une architecture à base de composants existe. De même un prototype d'extension de l'usine FRACTAL ADL transformant dynamiquement les nœuds de l'AST FRACTAL ADL est également réalisé. Le regroupement de ces deux prototypes va être réalisé à moyen terme.

Du point de vue de l'évaluation du coût de notre proposition, l'utilisation de nos deux opérateurs n'introduit aucun surcoût à l'exécution mais il impacte les phases de configuration et de reconfiguration en vérifiant les invariants d'architecture. En effet, ceux-ci ne définissent aucun intercepteur et par conséquent ils n'impactent pas le code métier des composants. Néanmoins, l'introduction de ces deux opérateurs nécessite une extension de la chaîne de chargement (composant Loader) de FRACTAL ADL. Cette extension réalise des modifications et des vérifications sur les nœuds de l'AST créés par le composant BasicLoader. Ces traitements introduisent un surcoût au chargement des composants mais la concision de la syntaxe des descriptions FRACTAL ADL utilisant nos opérateurs permet de réduire le temps de l'analyse syntaxique réalisée par le composant BasicLoader. Par manque de temps, une évaluation empirique du temps de chargement des composants n'a pas pu être réalisée pour évauler les surcoûts de notre proposition.

6.6 Travaux connexes

Cette section compare notre proposition avec les travaux existants sur la manipulation d'architectures pour la factorisation des motifs d'assemblage (*cf.* section 6.6.1), la vérification d'invariants d'architecture (*cf.* section 6.6.2) et l'intégration de nouvelles préoccupations (*cf.* section 6.6.3).

6.6.1 Description de motifs d'architecture

Comme nous avons pu le présenter dans la section 3.3.4, le langage de description d'architectures DARWIN intègre également un opérateur de factorisation des descriptions d'architecture [MDK94]. Cet opérateur, identifié par le mot-clé `forall`, permet d'effectuer une itération bornée sur un tableau d'objets préalablement déclaré avec le mot-clé `array`. Cet opérateur simplifie la description de structures récurrentes mais ses capacités sont limitées comparé aux opérateurs `foreach` et `assert` présentés dans ce chapitre. En particulier, l'opérateur `forall` défini dans DARWIN ne permet de réaliser que des itérations bornées sur des ensembles homogènes de composants. Notre opérateur `foreach` permet quant à lui d'itérer sur un ensemble d'éléments caractéristiques de l'architecture hétérogènes (par exemple, composants, interfaces) capturés à l'aide d'une requête FPATH. Notre opérateur `assert` garantit la validité du résultat d'une itération en exprimant un invariant d'architecture que l'architecte doit respecter lors de l'assemblage de l'application. Ce type de vérification n'est pas disponible dans l'ADL DARWIN et celui-ci ne permet donc pas de vérifier la validité des paramètres de l'opérateur `forall`.

6.6.2 Vérification d'invariants d'architecture

L'approche PLASTIK présentée dans la section 3.3.2 propose non seulement un langage de description d'architectures pour le modèle de composants OPENCOM, mais aussi un ensemble de mécanismes pour supporter les reconfigurations anticipées ou non d'une architecture à base de composants. Concernant les reconfigurations anticipées, PLASTIK propose un mécanisme basé sur des règles de type Événement–Condition–Action (ECA) permettant de détecter la variation d'un contexte et de modifier la configuration architecturale en fonction de la nouvelle situation. Sur ce point, l'approche proposée dans SAFRAN [DL06] propose un mécanisme similaire pour supporter la reconfiguration dynamique des architectures. En effet, dans SAFRAN, les variations du contexte sont détectées par le canevas WILDCAT tandis que les opérations de reconfigurations de l'architecture sont exprimées en utilisant le langage FSCRIPT [Dav05]. Par contre, SAFRAN ne propose pas de solution pour la prise en charge des reconfigurations non-anticipées. Dans PLASTIK, ce type de reconfiguration est contrôlé par la définition d'un invariant d'architecture qui est vérifié à l'exécution lorsque l'application est reconfigurée. Cette vérification garantit que toute reconfiguration *ad-hoc* de l'architecture respecte les invariants décrits par l'architecte. Notre opérateur `assert` permet de réaliser ces vérifications sur la cohérence de l'architecture d'une application lors de son déploiement et de son exécution. L'invariant d'architecture est exprimé dans notre proposition par une requête FPATH. Son interprétation lors du déploiement est réalisée par une extension de l'usine FRACTAL ADL et l'utilisation de notre implantation de FPATH permettant de naviguer dans les nœuds de l'AST FRACTAL ADL. L'interprétation à l'exécution de la requête est réalisée par une extension du contrôle des composants FRACTAL et l'utilisation de la configuration de FPATH permettant de naviguer dans une application en cours d'exécution.

6.6.3 Intégration de nouvelles préoccupations

Les canevas logiciel TRANSAT [BLLD06] et FIESTA [WLD06] proposent une approche pour faciliter l'intégration de nouvelles préoccupations dans une architecture à base de composants. Ces approches reposent sur la notion de patron d'architecture pour identifier une préoccupation à intégrer. Ce patron d'architecture se compose d'un masque de coupe, d'un nouveau plan à intégrer et d'un ensemble de règles d'intégration. Le masque de coupe identifie un emplacement d'intégration dans l'architecture. Le nouveau plan correspond à une description architecturale d'un ensemble de composants à intégrer dans l'architecture existante. Les règles d'intégration permettent de modifier l'architecture existante pour la mettre en relation avec le nouveau plan. L'approche présentée dans ce chapitre propose un formalisme pour factoriser et vérifier les constructions architecturales. Les opérateurs que nous définissons peuvent être rapprochés du

patron d'architecture présenté dans TRANSAT et FIESTA. En effet, le masque de coupe correspond à la requête FPATH alors que le nouveau plan représente la description architecturale à intégrer dans notre approche. Les éléments contenus dans l'opérateur `foreach` peuvent être assimilés aux règles d'intégration définies dans TRANSAT et FIESTA et notre opérateur `assert` permet de vérifier la validité de l'intégration de la préoccupation. Cette vérification est réalisée dans TRANSAT et FIESTA par l'algorithme d'intégration qui charge les règles d'intégration pour les appliquer. À la différence de ces approches, nous proposons un formalisme auto-contenu pour identifier une préoccupation à intégrer. Cependant, notre approche adresse la factorisation des descriptions d'architecture plutôt que l'intégration de nouvelles préoccupations.

6.7 Conclusion

Ce chapitre a présenté notre langage de description et de vérification de motifs d'architecture pour faire face à la complexification des descriptions d'architectures actuelles et améliorer la séparation des préoccupations dans les langages de description d'architectures. Afin d'améliorer le passage à l'échelle des descriptions d'architecture et de réduire les erreurs syntaxiques et sémantiques, nous avons défini deux nouveaux opérateurs de description et de vérification de motifs d'architecture pour le langage FRACTAL ADL. Ces opérateurs utilisent une version étendue du langage FPATH pour réaliser des requêtes d'interrogation sur le contenu de l'architecture. Le résultat de ces requêtes est ensuite exploité par les opérateurs pour générer un motif d'architecture (opérateur `foreach`) ou pour détecter une erreur d'assemblage (opérateur `assert`) ou de reconfiguration *ad-hoc* (par extension du contrôle des composants FRACTAL).

Grâce à ces opérateurs, il est non seulement possible de réduire drastiquement la taille des définitions FRACTAL ADL mais il est également possible d'identifier un certain nombre de motifs d'architecture génériques pouvant être facilement réutilisés dans différents contextes.

La réalisation de ces opérateurs se base sur une implantation à base de composants FRACTAL du langage FPATH. Cette implantation étend la version actuelle de FPATH en supportant non seulement la navigation dans les architectures disponibles à l'exécution mais aussi dans les descriptions de ces architectures réalisées avec le langage FRACTAL ADL. Grâce à cette extension de FRACTAL ADL, les descriptions disponibles dans le canevas logiciel GOTM peuvent être réduites et modularisées.

L'opérateur `foreach` peut également être utilisé pour réaliser du placement de composants. Étant donné qu'il est possible de combiner l'opérateur `foreach` avec les différentes constructions du langage FRACTAL ADL, le placement des composants FRACTAL avec le canevas de communication FRACTAL RMI peut être ainsi automatisé. Cette procédure consiste à définir une requête FPATH sélectionnant tous les composants à héberger sur un nœud déterminé et à surcharger leur définition pour introduire une balise `<virtual-node/>` spécifiant l'identifiant du nœud.

De plus, les deux implantations de FPATH que nous avons conçues peuvent être réutilisées dans le cadre de la programmation par aspects. En effet, les requêtes FPATH constituent une forme de langage de coupe pour les architectures à base de composants. Dès lors, il est possible d'utiliser FPATH dans un contexte d'application des aspects dans les architectures à composants (comme proposé dans l'approche FRACTAL ASPECT COMPONENTS [PSCD06]) afin d'offrir un langage dédié à la sélection des éléments caractéristiques d'une architecture logicielle.

All components become obsolete.

Murphy's Law

Chapitre 7

Le canevas GOTM pour la construction de services de transactions hautement adaptables

Sommaire

7.1 Introduction 124
7.2 Étude des points de variation 124
7.2.1 Adaptation du standard transactionnel 125
7.2.2 Adaptation du protocole de validation 125
7.2.3 Adaptation du modèle de transactions 126
7.2.4 Synthèse 126
7.3 Motivations 126
7.3.1 Flexibilité des services de transactions 126
7.3.2 Support hautement extensible pour l'adaptation 127
7.3.3 Granularité extrêmement fine de l'adaptation 127
7.4 Architecture du canevas GOTM 127
7.4.1 Modèle d'un service de transactions adaptable 127
7.4.2 Représentation dynamique d'un service de transactions 129
7.4.3 Architecture à base de composants d'un service de transactions 129
7.5 Éléments d'implantation du canevas GOTM 129
7.5.1 Implantation du motif de conception Adapter 131
7.5.2 Implantation des motifs de conception Factory et Prototype 132
7.5.3 Implantation du motif de conception Singleton 133
7.5.4 Implantation du motif de conception Strategy 134
7.5.5 Implantation du motif de conception State 135
7.5.6 Implantation du motif de conception Command 136
7.5.7 Implantation du motif de conception Publish-Subscribe 138
7.6 Scénarii d'adaptation de GOTM 138
7.6.1 Adaptation du protocole de validation de la transaction 139
7.6.2 Adaptation des participants d'une transaction 139
7.6.3 Adaptation d'un standard transactionnel 139
7.7 Travaux connexes 140
7.7.1 Canevas intergiciels 140
7.7.2 Aspects et motifs de conception 140

7.8 **Conclusion** 141

CE CHAPITRE PRÉSENTE GOTM, *notre canevas logiciel à base de composants pour la construction de services de transactions hautement adaptables. Ce canevas logiciel propose une architecture abstraite de services de transactions. Cette abstraction se repose sur l'identification de différents motifs de conception communs à tous les services de transactions. Ces motifs sont réifiés sous la forme de composants et peuvent être raffinés pour donner lieu à l'implantation de différents modèles de transactions, protocoles de validation ou standards transactionnels.* GOTM *propose ainsi une bibliothèque extensible de composants implantant les fonctions généralement requises lors du développement d'un service de transactions.*
Les composants primitifs de GOTM *utilisent notre modèle de programmation* FRACLET *tandis que les assemblages des motifs de conception exploitent notre extension de* FRACTAL ADL. *Grâce à ces deux outils et à l'identification des motifs de conception,* GOTM *fournit une infrastructure de services de transactions facilement extensible et hautement adaptable.*

7.1 Introduction

Les services de transactions existants présentent des architectures assez similaires [PSWL95, BP96, Mes03, Lit05]. Ces architectures sont fortement dirigées par les standards transactionnels que les services de transactions mettent en œuvre [Che99, OMG03, CCF+05a]. Généralement, un service de transactions comporte une partie statique appelée *service* et une partie dynamique appelée *transaction*. Ces deux parties remplissent deux fonctions bien distinctes et présentent par conséquent des architectures différentes.

Afin d'ouvrir l'architecture des services de transactions, nous utilisons différents motifs de conception qui ont été définis par le *Gang of Four* (GoF) [GHJV95]. Ces motifs de conception présentent un niveau d'abstraction suffisant pour représenter les grandes fonctions d'un service de transactions sans imposer une sémantique particulière. Chacun de ces motifs de conception peut donner lieu à différentes implantations répondant à des préoccupations particulières (par exemple, protocole de validation, type de participant, modèle de dépendance). L'implantation des composants élémentaires est réalisée en utilisant notre modèle de programmation FRACLET (présenté dans le chapitre 5) tandis que la description des motifs de conception de GOTM utilise notre langage de description et de vérification de motifs d'architecture (présenté dans le chapitre 6).

La suite de ce chapitre est organisée de la manière suivante. La section 7.2 identifie les points de variations d'un service de transactions face aux adaptations auxquelles il peut être confronté. La section 7.3 motive nos choix quant à la réalisation d'un canevas intergiciel pour la construction de services de transactions. La section 7.4 discute les aspects architecturaux de notre proposition. La section 7.5 présente quelques éléments d'implantation des points de variations de services de transactions construits avec GOTM. La section 7.6 illustre l'adaptation de différents aspects du service de transactions en situant les points d'impacts des modifications qu'elle représente. La section 7.7 situe notre proposition par rapport aux travaux connexes et la section 7.8 conclut ce chapitre.

7.2 Étude des points de variation

Cette section reprend les axes d'évolution que nous avons identifiés dans le chapitre 1 et positionne les points de variation qu'ils représentent dans l'architecture d'un service de transactions.

7.2.1 Adaptation du standard transactionnel

Les standards transactionnels assurent la compatibilité d'une application requiérant un support transactionnel avec le service de transactions assurant ce support. La nature et la forme de ces standards transactionnels reposent sur différentes technologies de mise en œuvre selon le contexte d'utilisation des services de transactions. Par exemple, le standard *Object Transaction Service* (OTS) [Che99] repose sur la technologie CORBA tandis que le standard *Web Services Atomic Transaction* (WS-AT) [CCF+05a] s'appuie sur les *Web Services*. Face à cette grande disparité de technologies, certains éléments peuvent néanmoins être isolés et abstraits dans ces standards transactionnels. Tout d'abord, le protocole de validation des transactions peut être le même pour plusieurs services de transactions puisque la plupart des standards transactionnels utilisent le protocole de validation en deux phases (2PC). Ensuite, le modèle de transactions peut être également commun à un ensemble de standards dans la mesure où les standards OTS et WS-AT reposent sur un modèle de transactions plat. Enfin, les fonctionnalités liées à la gestion des transactions (par exemple, la création d'une instance, l'activation d'une transaction, la gestion de l'instance courante) sont également prises en compte par chacun des standards transactionnels existants.

Dès lors, le point de variation que nous identifions parmi ces standards concerne l'adaptation des interfaces des différents standards transactionnels vers les abstractions des fonctionnalités. Cette adaptation adresse non seulement les interactions entre l'application et le service de transactions mais également les interactions entre les transactions et les participants qui y sont associés. Il est donc nécessaire d'identifier un mécanisme d'adaptation d'un ensemble d'éléments génériques aux différentes interfaces imposées par un standard transactionnel. Ce mécanisme d'adaptation impacte les interfaces du service de transactions et celles des transactions afin de supporter la diversité des participants pouvant être impliqués dans l'exécution d'une transaction.

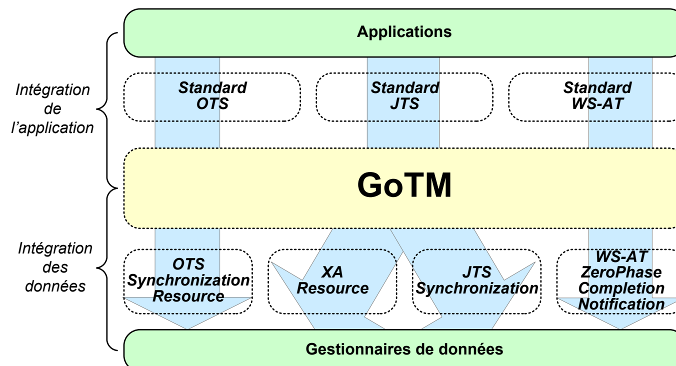


FIG. 7.1: Adaptation des standards transactionnels dans GoTM.

7.2.2 Adaptation du protocole de validation

Le protocole de validation d'une transaction est un algorithme plus ou moins complexe permettant d'assurer un consensus dans la décision de l'issue d'une transaction. Le protocole de validation le plus connu est le protocole de validation en deux phases (2PC) dont l'algorithme supporte la coordination d'un ensemble de participants distribués [BVVV05]. Cependant, le protocole 2PC peut présenter un certain nombre d'inconvénients lorsqu'il est appliqué dans des contextes d'exécution inadaptés. En particulier, ce protocole est incapable de prendre en compte les aspects de connexion/déconnexion des participants s'exécutant dans une topologie de réseau mobile [SARA04]. Il est donc nécessaire de développer différents algorithmes selon les particularités du contexte d'exécution d'une transaction afin d'optimiser l'adéquation du protocole de validation au contexte d'exécution. Par exemple, les protocoles 2PC hiérarchiques sont adaptés

aux réseaux de grande échelle tandis que les protocoles 2PC mobiles sont adaptés aux réseaux ubiquitaires.

Ce support de la variation du protocole de validation dans le service de transactions repose donc sur la définition d'une abstraction suffisante de la notion de protocole de validation d'une transaction afin de pouvoir implanter différents algorithmes à partir de cette abstraction. Ce point de variation impacte la transaction qui s'exécute dans un contexte donné et à laquelle un certain nombre de participants sont associés.

7.2.3 Adaptation du modèle de transactions

Le modèle de transactions spécifie le degré d'isolation qu'une transaction doit imposer à ses participants. Ce degré doit garantir que l'abandon d'une transaction n'impacte pas néfastement l'application (par exemple, rupture des invariants applicatifs). Cependant, la garantie de cette propriété implique un ensemble de contraintes fortes sur l'application (par exemple, verrouillage des données en cours de modification) qui peuvent nuire à son bon fonctionnement dans certains cas. Dès lors, les modèles de transactions se sont diversifiés pour répondre à la diversité des besoins applicatifs afin de fournir un degré d'isolation approprié. Par exemple, le modèle de transactions *Split/Join* a répondu aux besoins des applications de Conception Assistées par Ordinateur (CAO) dont les interactions transactionnelles pouvaient s'étaler sur plusieurs jours. Face à la diversité des besoins applicatifs en terme de support transactionnel, le service de transactions doit être en mesure d'adapter le modèle de transactions qu'il applique afin de ne pas impacter néfastement l'application.

Cette adaptation du modèle de transactions requiert de synthétiser les caractérisques d'un modèle de transactions afin qu'un service de transactions ne soit pas spécifiquement lié à un modèle de transactions donné. Ce point de variation impacte principalement la transaction dont le comportement doit être en mesure de varier selon le modèle de transactions qui lui est associé.

7.2.4 Synthèse

L'adaptabilité d'un service de transactions repose sur la prise en compte de nombreux paramètres qui peuvent être indépendants les uns des autres. En effet, si le choix d'un standard transactionnel repose sur un choix technologique, la sélection du protocole de validation dépend essentiellement du contexte d'exécution des transactions alors que la pertinence du modèle de transactions dépend du type d'application à laquelle il peut être associé.

De plus, ces différents points de variation impactent différents niveaux du service de transactions. Ainsi, la variation du standard transactionnel impacte principalement le service de transactions alors que la variation du protocole de validation ou du modèle de transactions concerne plus particulièrement la transaction. Ces niveaux de variation peuvent impliquer des granularités d'adaptations différentes.

Par conséquent, l'adaptation d'un service de transactions nécessite non seulement un mécanisme extensible au niveau du service et des transactions pour supporter les différentes stratégies associées à chaque point de variation mais également une granularité adéquate à l'adaptation des différents niveaux.

7.3 Motivations

Dans cette section, nous motivons nos choix liés à la réalisation du canevas intergiciel GoTM.

7.3.1 Flexibilité des services de transactions

Face à la grande diversité des éléments constituant un service de transactions, nous choisissons de développer un canevas intergiciel à base de composants pour la construction des services de transactions. Cette approche offre l'avantage d'être suffisamment ouverte pour ne pas nous

limiter aux problèmes cités précédemment mais pouvoir prendre en compte de nouveaux aspects pouvant apparaître lors de l'évolution du domaine transactionnel.

Ce canevas intergiciel se présente sous la forme d'une bibliothèque de composants logiciels pouvant être assemblés pour réaliser une instance particulière d'un service de transactions. Dès lors, chaque type de composant de cette bibliothèque représente un point de variation potentiel du service de transactions et les stratégies envisageables pour un point de variation correspondent à différentes implantations d'un type de composant de la bibliothèque.

7.3.2 Support hautement extensible pour l'adaptation

Afin de contrôler les points de variation d'un service de transaction, nous proposons d'exploiter l'extensibilité de motifs de conception existants [GHJV95]. Les motifs de conception ont été définis pour fournir une solution simple et élégante à des problèmes généralement rencontrés dans le développement d'applications orientées objets. Or, il s'avère que l'application de motifs de conception offre des points de variations facilement extensibles et réutilisables.

Dès lors, nous souhaiterions bénéficier de cette capacité d'adaptation offerte par les motifs de conception pour modéliser les points de variations d'un service de transactions. Dans la mesure où notre canevas intergiciel est réalisé en utilisant le paradigme de composant, nous proposons de réifier la structure des motifs de conception que nous appliquons au niveau de l'assemblage des composants afin de pouvoir utiliser le langage de description d'architecture comme un mécanisme d'adaptation du service de transactions.

7.3.3 Granularité extrêmement fine de l'adaptation

Pour supporter les différents niveaux d'adaptation, nous proposons de réduire la granularité d'adaptation à son maximum afin de pouvoir adapter tous les éléments caractéristiques d'une transaction comme ceux qui constituent le service auquel elle est associée. Notre objectif est de modéliser chaque caractéristique d'une transaction ou d'un service sous la forme d'un composant en cherchant à maximiser la réutilisation des composants. Pour ce faire, nous appliquons le paradigme de composant à tous les niveaux du service de transactions que ce soit lors de sa conception ou lors de l'exécution. Par conséquent, le service est un composant gérant un ensemble de composants de type transaction. Chaque transaction réifie ses fonctions telles le protocole de validation ou le modèle de transactions sous forme de composants. Enfin, les états supportés par un modèle de transactions particulier sont eux-mêmes représentés par des composants.

Ainsi, nous ne souhaitons pas supporter les différentes stratégies en augmentant la taille de la bibliothèque de composants mais nous préférons raffiner autant que possible le contenu de ces composants afin de décrire les stratégies à partir de composants réutilisables plutôt que de les programmer. Ce choix nous permet de modifier finement les caractéristiques d'une stratégie afin d'optimiser l'adéquation de la stratégie aux besoins de l'adaptation.

7.4 Architecture du canevas GOTM

Dans cette section, nous présentons l'architecture globale d'un service de transactions réalisable à partir du canevas intergiciel GOTM. Ce service repose sur un modèle extensible dont les éléments constitutifs représentent des motifs de conception. Chaque motif de conception correspond à un point de variation possible du service de transactions.

7.4.1 Modèle d'un service de transactions adaptable

La figure 7.2 présente un modèle extensible d'un service de transactions adaptable. Ce modèle comporte deux parties :

– Le Service représente la dimension statique du service de transactions,
– La Transaction est une représentation de la dynamique du service.

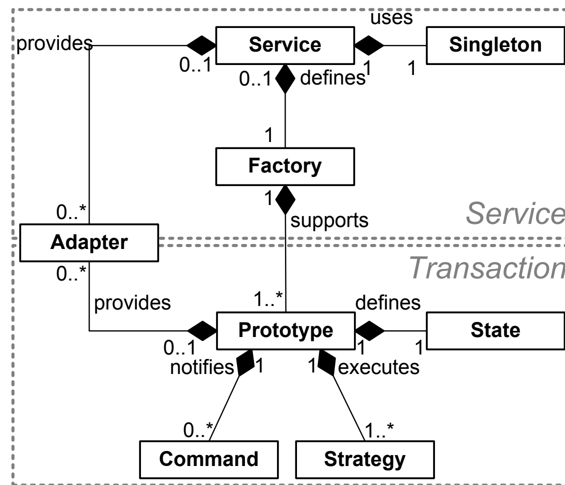


FIG. 7.2: Modèle d'un service de transactions adaptable.

Le Service utilise différents motifs de conception Adapter pour contrôler la variabilité des standards transactionnels. Chaque motif résoud le problème d'incompatibilité entre un standard transactionnel et les composants de la bibliothèque GOTM en isolant la conversion des interfaces sous la forme de composants fournissant les interfaces d'un standard et requiérant celles de notre bibliothèque. La composition des motifs de conception Adapter permet quant à elle de résoudre les problèmes d'interopérabilité du support transactionnel dans le cadre de systèmes applicatifs hétérogènes comme nous le détaillons dans le chapitre 10.

Il utilise également le motif Singleton pour gérer les instances de transactions actives et le motif Factory pour créer de nouvelles instances de transactions. Le motif Singleton garantit qu'une instance de classe n'est présente qu'en un seul exemplaire. Dans GOTM, il nous permet de partager l'instance d'une transaction entre les différents éléments applicatifs qui peuvent accéder au service de transactions. Le motif Factory offre une interface pour créer une famille d'objets de nature équivalente sans avoir à spécifier explicitement leur forme. Dans GOTM, ce motif nous permet d'offrir une abstraction de la création d'une transaction afin qu'un service de transactions soit en mesure de créer différents types de transactions.

Les type des instances de transactions créées par le motif Factory et supportées par le service de transaction est déterminé par les différents motifs Prototype qui peuvent lui être associé. Dans le chapitre 11, nous utilisons ce point de variation pour construire un service de transactions supportant différents protocoles de validation. Nous utilisons la notion de modèle (*template*) de composants pour réaliser ce motif de conception et cloner plusieurs instances de transactions à partir d'un type de composant transaction. Ainsi, le Prototype regroupe les motifs de conception Adapter, Command, Strategy et State qui représentent une abstraction de chaque caractéristique d'un type de transaction.

Le motif Adapter contrôle la variabilité du standard transactionnel en ce qui concerne les interfaces fournies par la transaction. Le motif Command supporte la diversité des participants qui peuvent être associés à une transaction. Ce motif est utilisé en complément du motif Adapter pour assurer la compatibilité de la transaction vers les participants.

Le motif Strategy définit le protocole de validation que la transaction doit utiliser. Ce motif de conception définit l'algorithme à appliquer pour valider ou annuler une transaction. Enfin, le motif State décrit le modèle de transactions appliqué. En modélisant le modèle de transactions sous la forme d'un automate présent à l'exécution, la transaction peut contrôler son comportement en se basant sur le motif State.

7.4.2 Représentation dynamique d'un service de transactions

La figure 7.3 présente une vue dynamique d'un service de transactions construit avec les motifs de conception que nous avons identifiés. Cette vue dynamique nous permet d'identifier les interactions existantes entre les différents motifs que ce soit au niveau du service de transactions ou au niveau des transactions.

La séquence décrite dans ce diagramme présente le cas d'une application souhaitant modifier une donnée dans le cadre d'une transaction. La donnée modifiée est matérialisée par un participant enregistré dans la transaction. Cette transaction est débutée à l'initiative de l'application. Le participant est ensuite enregistré dans la transaction avant que la donnée ne soit modifiée. Dans cet exemple, l'application utilise un mode de programmation indirecte (présenté dans la section 2.2.6 du chapitre 2) en accédant systématiquement à la transaction courante du service.

Lors de l'activation de la transaction, le motif Adapter récupère l'instance de la transaction courante via le motif Singleton. Si aucune transaction n'est alors disponible, le motif Singleton utilise les motifs Factory et Prototype pour en créer une nouvelle instance.

Lors de l'enregistrement d'un participant dans une transaction, le motif Command de la transaction mémorise la référence du participant puis le notifie de sa participation à la transaction.

Lors de la validation de la transaction, le motif Strategy de la transaction exécute l'algorithme du protocole de validation. Pour ce faire, le motif Strategy modifie l'état de la transaction en utilisant le motif State pour donner lieu à une notification du motif de conception Command afin que l'évolution de l'état de la transaction soit prise en compte par les participants de la transaction.

7.4.3 Architecture à base de composants d'un service de transactions

Les figures 7.4 et 7.5 présentent une architecture à base de composants du service de transactions et des transactions supportées par le service. Chaque composant correspond à l'un des motifs de conception que nous avons précédemment identifiés. Ainsi, la figure 7.4 se concentre sur le service de transactions. Cette architecture met en évidence les connexions existantes entre le motif Adapter et les motifs Singleton et Factory (non illustré dans la figure 7.3) mais aussi la liaison de Factory et de Prototype.

La figure 7.5 modélise l'architecture d'une transaction et présente l'articulation des motifs de conception qui la composent. Cette architecture correspond au contenu du composant Prototype géré par le service de transactions. Ainsi, le motif Adapter fournit une façade des motifs Strategy, State et Command adaptée à un standard transactionnel particulier. La propagation des messages échangés entre les motifs Strategy, State et Command est réalisée par le motif Publish-Subscribe que nous introduisons au niveau de l'architecture pour fournir un mode de communication par messages.

Cette architecture abstraite d'un service de transactions est indépendante des stratégies envisageables pour chaque point de variation que nous avons identifiés. La réalisation de ces stratégies dépend par conséquent du contenu de chacun de ces composants. Ainsi, le contenu de ces composants exploite l'extensibilité des motifs de conception qu'ils représentent pour fournir des implantations différentes représentant les adaptations possibles du service de transactions.

7.5 Éléments d'implantation du canevas GOTM

Dans cette section, nous présentons le contenu des différents motifs de conception que nous avons identifiés précédemment. Après avoir présenté un exemple d'architecture du motif, nous étudions les extensions applicables au motif pour supporter d'autres stratégies d'implantation.

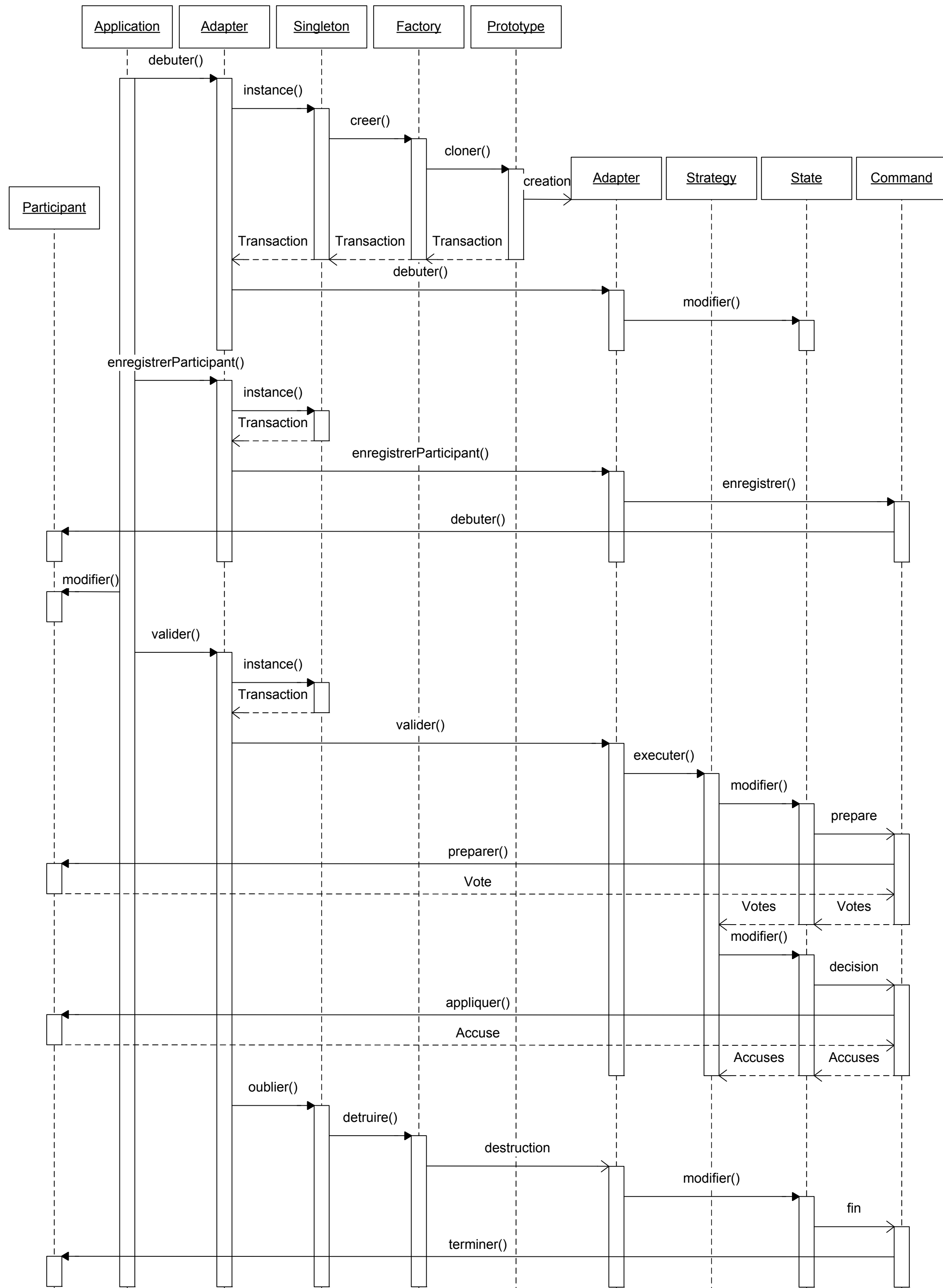


FIG. 7.3: Diagramme UML de séquences d'un service de transactions.

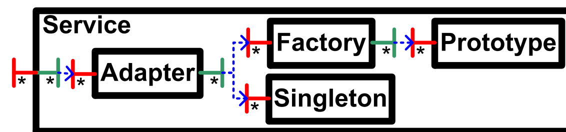


FIG. 7.4: Architecture abstraite du service de transactions.

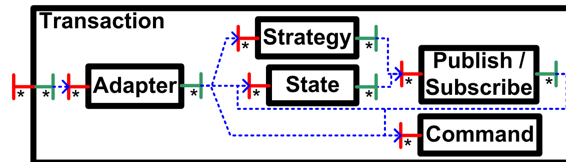


FIG. 7.5: Architecture abstraite de la transaction.

7.5.1 Implantation du motif de conception Adapter

Définition. Le motif de conception Adapter nous permet de supporter la variabilité des standards transactionnels en assurant la conversion des interfaces du standard transactionnel vers les interfaces de GOTM selon la définition suivante :

> *Convert the interfaces of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.*
>
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Application. La figure 7.6 présente la définition d'un service de transactions respectant le standard transactionnel JTS [Che99]. Pour ce faire, une transaction générique (représentée par les patrons Strategy, State et Publish-Subscribe) est adaptée au standard JTS avec un composant JTS Adapter. Ce composant est donc placé en amont de la transaction générique pour convertir les interfaces requises par le standard (ses interfaces fournies) vers les interfaces supportées par le canevas GOTM (ses interfaces clientes). En utilisant le motif Adapter, le service de transactions peut être introduit de manière transparente dans une plate-forme utilisant un standard transactionnel particulier.

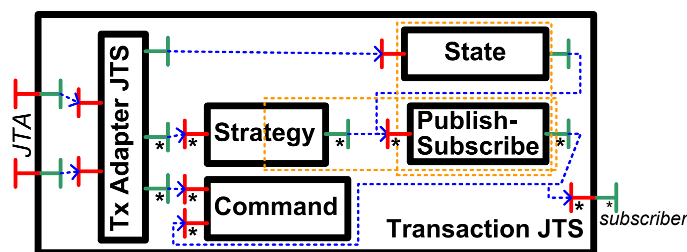


FIG. 7.6: Mise en œuvre du motif de conception Adapter dans GOTM.

Extensions. Les extensions possibles du motif de conception Adapter concernent le support de nouveaux standards de transactions. Pour exploiter ce point de variation, il suffit de développer les composants XXX Service Adapter, XXX Transaction Adapter et éventuellement XXX Command pour chaque standard transactionnel XXX à implanter.

En outre, une expérience de construction d'un service de transactions utilisant plusieurs composants Adapter pour supporter plusieurs standards simultanément est présentée dans le chapitre 10.

7.5.2 Implantation des motifs de conception Factory et Prototype

Définitions. Le motif de conception Factory offre un support uniforme pour la création des instances de transactions dans GOTM en nous basant sur la définition suivante :

> *Provide an interface for creating families of related or dependent objects without specifying their concrete class.*
>
> ---
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Le motif de conception Prototype nous permet de modéliser le type d'une transaction et de créer des instances à partir de ce type d'après la définition suivante :

> *Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.*
>
> ---
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Application. Lors de leur intégration dans GOTM, les motifs de conception Factory et Prototype ont offert des solutions intéressantes pour la gestion des transactions. En effet, l'utilisation du motif de conception Prototype permet de réifier à l'exécution le type des transactions gérées par le service de transactions. Il est ainsi possible de modifier dynamiquement la nature des transactions supportées par le service afin d'adapter son comportement à un besoin applicatif donné. Le modèle de composants FRACTAL facilite la réalisation du motif de conception Prototype en définissant la notion de composant *template*. Un composant *template* est un composant applicatif doté d'un contrôleur particulier. Ce contrôleur fournit une interface *factory* et une méthode `newInstance()` permettant de cloner le composant *template* en un composant applicatif.

Le motif de conception Factory offre quant à lui la possibilité d'intégrer différentes préoccupations à la gestion des instances. Il est ainsi possible d'intégrer un mécanisme de gestion de cache afin de recycler les instances des transactions et réduire le coût de création des transactions. De la même façon, il est possible d'intégrer un mécanisme de gestion et de manipulation des instances de transactions actives. Un exemple d'architecture de fabrique utilisée dans GOTM est ainsi présenté dans la figure 7.7. Les composants Basic Factory et Prototype représentent les motifs de conception Factory et Prototype. Ces composants sont ensuite utilisés par les composants Cache Factory et Cache Pool afin d'introduire un mécanisme de cache au niveau de la gestion des instances. Ce mécanisme de cache utilise un contrôleur particulier permettant de recycler les instances existantes. Les composants Pool Factory et Instance Pool permettent de lister les instances créées par la fabrique d'instances.

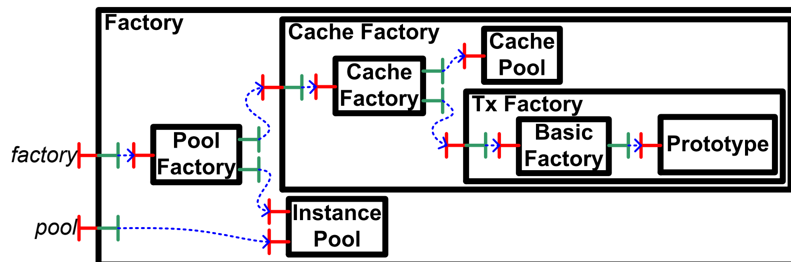


FIG. 7.7: Mise en œuvre des motifs de conception Factory et Prototype dans GOTM.

Extensions. Les extensions possibles des motifs Factory et Prototype adressent tant les aspects de performances du processus de création des transactions que le contrôle de l'allocation des transactions. En effet, il est possible de définir une politique permettant de contrôler finement le nombre de transactions instanciées par *thread* ou par *zone* appelante de façon analogue aux politiques définies pour le motif de conception Singleton.

7.5.3 Implantation du motif de conception Singleton

Définition. Le motif de conception Singleton permet de maintenir les instances de transactions en cours d'exécution dans GOTM en nous inspirant de la définition suivante :

> *Ensure a class only has one instance, and provide a global point to access to it.*
>
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Application. Lors de son intégration dans GOTM, nous avons mis en œuvre différentes politiques de gestion du singleton. Ces politiques sont isolées dans des composants et leur configuration est réalisée dans les descripteurs d'assemblage du service. Par conséquent, la réalisation des différentes politiques du composant Singleton nécessite la définition préalable d'une interface *Singleton* fournie par toutes les politiques du motif de conception Singleton. Le tableau 7.1 résume l'ensemble des politiques de gestion du singleton réalisées par des composants FRACTAL fournissant l'interface *Singleton*. À la politique traditionnelle Single Policy de gestion du singleton, nous avons ajouté une politique Thread Policy associant une instance de transactions au *thread* appelant. La politique Area Policy utilise la notion de *zone* définie par le projet THREAD MANAGEMENT FRAMEWORK. Les zones permettent un découpage spatial d'une application en fonction de l'organisation des paquetages ou des classes. Ce découpage est réalisé via un descripteur de configuration associant les zones et les entités de l'application. Lors d'un accès à l'instance maintenue par le singleton, la zone associée à l'entité appelante est inférée et le composant Area Policy retourne alors la transaction affectée à cette zone. Ces nouvelles politiques permettent l'exécution concurrentielle de plusieurs transactions par un même service.

Libellé	Sémantique
Single Policy	Une seule instance est disponible dans le système.
Thread Policy	Une seule instance est disponible par *thread* appelant.
Area Policy	Une seule instance est disponible par *zone* appelante.

TAB. 7.1: Politiques du composant Singleton.

La figure 7.8 illustre l'intégration de la politique Thread Policy du composant Singleton dans l'architecture du service de transactions. Cette politique partage l'instance du composant Factory avec celle définie par le composant Transaction Service. La liste des transactions disponibles est réifiée sous la forme d'une interface cliente collection disponible sur le composant Thread Policy.

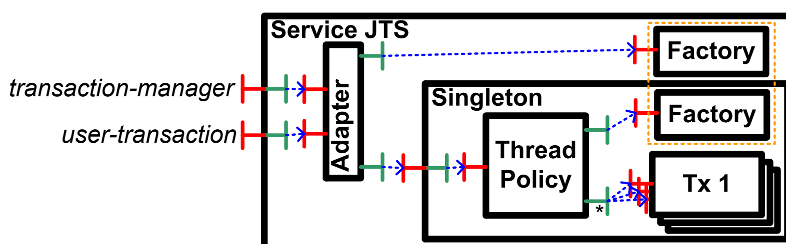


FIG. 7.8: Mise en œuvre du motif de conception Singleton dans GOTM.

Extensions. Les extensions envisageables pour le motif de conception Singleton adressent la définition de nouvelles politiques de gestion du singleton. Ces nouvelles politiques permettent de changer l'association d'une transaction avec son contexte d'exécution.

7.5.4 Implantation du motif de conception Strategy

Définition. Le motif de conception Strategy est appliqué dans GOTM non seulement pour supporter différents protocoles de validation mais aussi pour offrir un moyen uniforme d'exécuter les différents algorithmes de synchronisation que ce soit pour les procédures de validation (*commit*) ou d'abandon (*abort*) selon la définition suivante :

> *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*
>
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Application. La figure 7.9 présente l'utilisation du motif de conception Strategy dans GOTM. Les stratégies disponibles correspondent à l'exportation des interfaces *execute* des composants Vote, Commit et Abort pour réaliser respectivement les algorithmes de validation en 2 phases, de validation en 1 phase et d'abandon d'une transaction. Les liaisons entre ces composants et le composant Publish-Subscribe correspondent au type de propagation des messages de l'algorithme (synchrone ou asynchrone) tandis que les liaisons avec les composants Log correspondent au type de journalisation à employer (immédiat ou non). Les détails liés à la réalisation du protocole de validation via des composants de fine granularité sont présentés dans les chapitres 8 et 11.

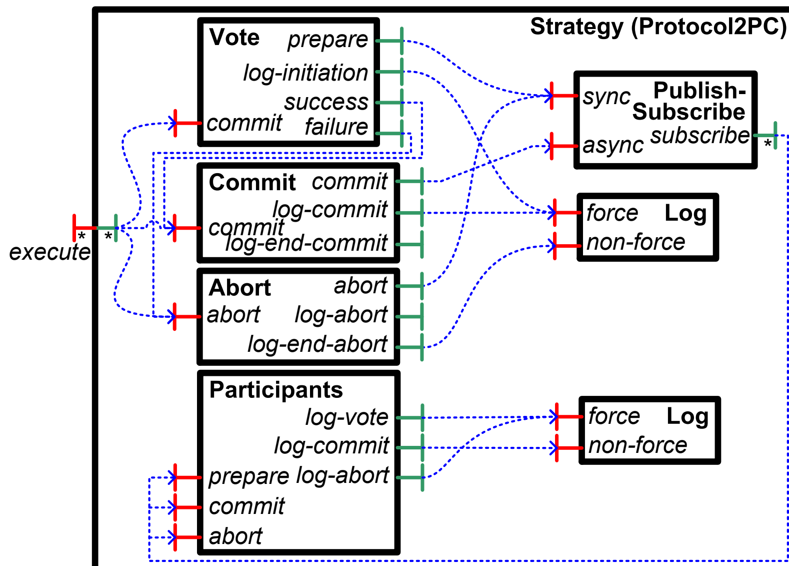


FIG. 7.9: Mise en œuvre du motif de conception Strategy dans GOTM.

Le motif de conception Strategy est également appliqué au niveau du gestionnaire de participants (voir la figure 7.12) et du bus de communication (voir la figure 7.13) pour supporter les différentes politiques de propagation.

Extensions. Les extensions possibles du motif de conception Strategy dans GOTM concernent la prise en compte de nouveaux protocoles de validation et de leurs optimisations. D'ailleurs, une étude des optimisations possibles du protocole de validation en 2 phases est présentée dans le chapitre 11.

7.5.5 Implantation du motif de conception State

Définition. Le motif de conception State est appliqué dans GOTM pour identifier les différents états d'une transaction et leurs associés des comportements différents en nous inspirant de la définition suivante :

> *Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.*
>
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Application. La figure 7.10 présente la modélisation des états d'une transaction plate utilisant un protocole de validation à deux phases. Les connexions entre le composant State Manager et l'ensemble des états accessibles ne sont pas modélisées par souci de clarté. Les dépendances entre les différents états représentent les transitions et le nom associé à la liaison cliente représente l'événement déclencheur de la transition. Lorsqu'un événement est émis, le composant Atomic Transaction State vérifie que pour l'état courant, il existe une transition correspondant à l'événement émis. Si une telle transition existe l'état courant est mis à jour, et dans le cas contraire une exception est levée.

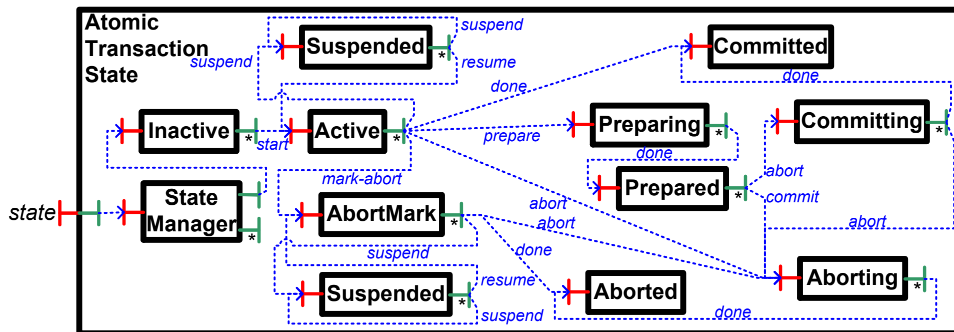


FIG. 7.10: Mise en œuvre du motif de conception State dans GOTM.

Le listing 7.1 illustre la réutilisation des composants de GOTM. Le composant `StateMachine` raffine la définition `StateManagerAutoBind` générée par FRACLET (ligne 1) pour introduire des invariants d'architecture liés à la définition d'une machine à états. La première assertion garantit que la machine à états est configurée avec un état initial (lignes 2–3). La deuxième assertion garantit qu'il existe au moins un état final dans la machine à états (lignes 4–5). La troisième assertion vérifie que tout état non final doit être connecté à au moins un autre état (lignes 6–7). Cette définition est ensuite étendue par la définition `AtomicTransactionState` afin de décrire la machine à états d'une transaction plate (ligne 10). En particulier, la définition `StateImpl` représente le composant primitif utilisé pour représenter un état (lignes 11–13). L'implantation de ce composant est générique et peut être utilisée pour composer n'importe quel automate. L'état initial est déterminé par la liaison `initial` du composant `client` hérité de la définition `StateManagerAutoBind` (ligne 15). L'état final est paramétré par un attribut booléen de la définition `StateImpl`. Le nom des liaisons clientes détermine les événements déclencheurs des transitions d'états (lignes 16–18)

Extensions. Une réalisation sous forme de composants de ce motif de conception nous permet de gérer la description des états et des transitions supportées par une transaction. Grâce à ce composant, il nous est possible de décrire des comportements transactionnels plus élaborés tels que la prise en compte des transactions avancées. En particulier, nous pouvons faire évoluer le composant AtomicTransactionState pour y intégrer de nouveaux états pour la compensation d'une transaction [Nem04].

```
1 <definition name="StateMachine" extends="StateManagerAutoBind">
2   <assert condition="bound(child::client::binding::initial)"
3           message="The state machine should define one initial state."/>
4   <assert condition="count(child::*::attribute::final[value(.)==true])>0)"
5           message="The state machine should define at least one final state."/>
6   <assert message="The state machine can not define well states."
7    condition="count(child::*[attribute::final[value(.)==false]]::binding::state-*)>0"/>
8 </definition>

10 <definition name="AtomicTransactionState" extends="StateMachine">
11   <component name="inactive"  definition="StateImpl(true)">
12   <component name="active"    definition="StateImpl(false)">
13   <component name="suspended" definition="StateImpl(false)">
14   <!-- [...] -->
15   <binding client="client.initial"         server="inactive.state"/>
16   <binding client="inactive.state-start"   server="active.state"/>
17   <binding client="active.state-suspend"   server="suspended.state"/>
18   <binding client="suspended.state-resume" server="active.state"/>
19   <!-- [...] -->
20 </definition>
```

LST. 7.1: Description FRACTAL ADL des composants StateMachine et State.

7.5.6 Implantation du motif de conception Command

Définition. Le motif de conception Command est appliqué dans GOTM pour notifier les participants des modifications de l'état de la transaction à laquelle ils sont associés. Pour ce faire, les modifications de l'état de la transaction sont réifiées sous la forme de requêtes en nous appuyant sur la définition suivante :

> *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.*
>
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Application. Dans le contexte du canevas logiciel GOTM, le motif de conception Command permet de définir les commandes réalisables par la transaction sur ces participants. Ainsi, la figure 7.11 présente les commandes disponibles sur les participants de type *synchronization*. Ces commandes permettent de notifier les participants de la transaction avant et après la validation de la transaction. Chacune de ces commandes partage le composant State de la transaction afin de pouvoir consulter l'état courant de la transaction.

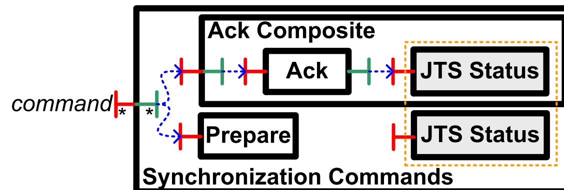


FIG. 7.11: Définition du composant regroupant les commandes de type `Synchronization`.

Le listing 7.2 présente la définition FRACTAL ADL du composant `SynchronizationCommands` supportant les commandes `prepare` (ligne 3) et `ack` (ligne 4) correspondant aux actions applicables sur les participants de type `Synchronization`.

Le listing 7.3 présente l'interface `Command` (lignes 1–4) utilisée pour réaliser le motif de conception Command et deux composants `Prepare` (lignes 5–9) et `Ack` (lignes 10–15) définis pour supporter les opérations de l'interface `Synchronization` du standard JTS. La réalisation du composant Synchronization Commands consiste à implanter l'interface `execute()` de l'interface et à décrire l'assemblage des commandes.

```
1 <definition name="SynchronizationCommands"
2          extends="AutoExport(command,Command),AutoSharing(status,AtomicTransactionState)">
3   <component name="prepare" definition="Prepare"/>
4   <component name="ack"     definition="AckComposite"/>
5 </definition>
```

LST. 7.2: Description FRACTAL ADL du composant SynchronizationCommands.

```
1 /** @provides */
2 public interface Command {
3   Object execute(Map context, Object participant) throws Exception;
4 }
5 public class Prepare implements Command {
6   public Object execute(Map context, Object participant) throws Exception {
7     ((Synchronization)participant).beforeCompletion();
8     return null;
9 } }
10 public class Ack implements Command {
11   /* @requires */ protected Status status;
12   public Object execute(Map context, Object participant) throws Exception {
13     ((Synchronization)participant).afterCompletion(status.getStatus());
14     return null;
15 } }
```

LST. 7.3: Implantation des commandes de *Synchronization* dans GOTM.

Pour réaliser l'intégration du motif de conception Command dans GOTM, nous souhaitons mettre en œuvre différentes politiques d'exécution des commandes définies dans le composant Command telles que la notification séquentielle ou simultanée des participants d'une transaction. La figure 7.12 compose un gestionnaire de participant (composant Participant Pool), des politiques de notification (composants Sequence Notify et Parallel Notify) et les actions applicables sur les participants (composant Synchronization Commands).

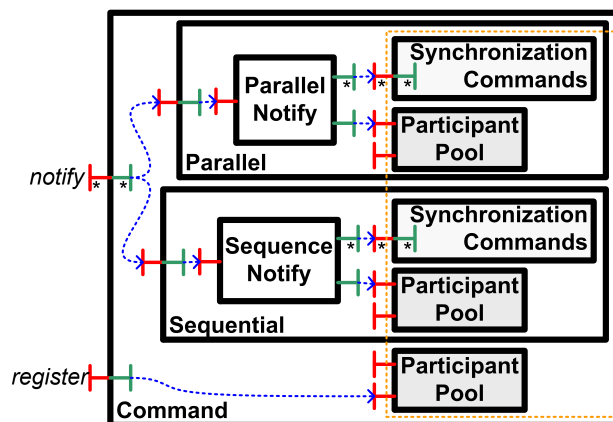


FIG. 7.12: Mise en œuvre du motif de conception Command dans GOTM.

Dans la figure 7.12, les composants Sequence Notify et Parallel Notify réalisent également le motif de conception Command dans la mesure où les deux politiques d'application d'une commande implémentent une même interface `Notify` avec deux stratégies différentes.

Extensions. L'utilisation du motif de conception Command pour la gestion des participants nous permet d'intégrer différents types de participants (par exemple, *Resource*, *Synchronization*) pour que ceux-ci soient pris en compte de manière uniforme par la transaction. Les extensions possibles de ce motif de conception adressent non seulement le support de différents types

de participants mais aussi le support des verrous d'une transaction afin d'en automatiser le dévérrouillage lors de l'exécution de l'algorithme de validation en 2 phases.

7.5.7 Implantation du motif de conception Publish-Subscribe

Définition. Le motif de conception Publish-Subscribe est appliqué dans GOTM pour assurer la propagation de la notification des motification de l'état de la transaction vers les différents types de participants supportés par la transaction en utilisant la définition suivante :

> *Define one-to-many dependency between objects so that when one object changes state, al its dependents are notified and updated automatically.*
>
> *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]

Application. Lors de l'intégration de ce composant dans GOTM, nous avons développé le motif de conception Publish-Subscribe sous la forme d'un bus d'intégration capable d'intégrer des politiques de notification synchrone et asynchrone, et un mécanisme de contrôle de notification [Esk99]. Ces mécanismes sont présentés dans la figure 7.13. En particulier, les composants Synchronous Publisher et Asynchronous Publisher réalisent les fonctions de notification synchrone et asynchrone. En amont de ces composants est placé un composant State Checker dont le rôle est d'intercepter les requêtes de notification afin de s'assurer que ces requêtes sont conformes avec l'état courant de la transaction. Par exemple, une transaction ne peut pas évoluer vers l'état *committed* si celle-ci se trouve dans l'état *marked for rollback*.

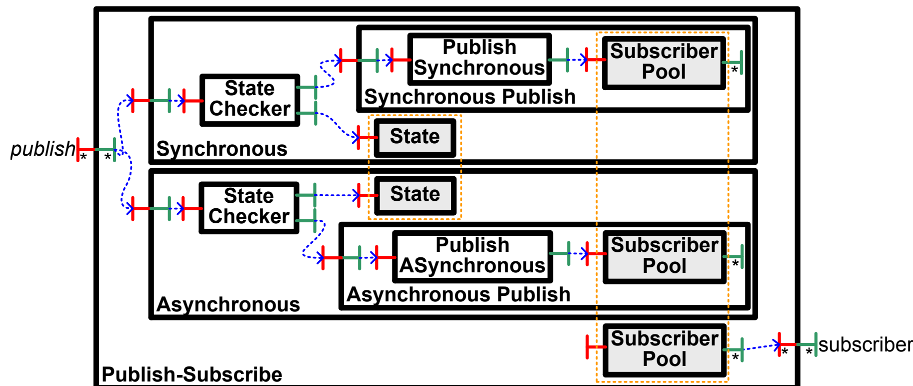


FIG. 7.13: Mise en œuvre du motif de conception Publish-Subscribe dans GOTM.

Extensions. Le composant Publish-Subscribe présenté ici est suffisamment flexible pour autoriser différentes politiques de notification (notification synchrone, asynchrone libre, asynchrone contrôlée, etc.). De plus, il est également possible d'étendre ce composant pour ajouter des fonctionnalités supplémentaires (contrôle des événements, trace des événements, etc.) sans impacter le reste du système. Grâce à ce composant, il est possible de configurer la transaction afin de favoriser la performance ou la sûreté d'exécution selon le contexte dans lequel le service de transactions est déployé.

7.6 Scénarii d'adaptation de GOTM

Dans cette section, nous présentons divers scénarii d'adaption d'un service de transactions GOTM afin d'illustrer l'impact des modifications que représentent ces adaptations sur l'architecture d'un service de transactions.

7.6.1 Adaptation du protocole de validation de la transaction

L'adaptation du protocole de validation consiste à modifier l'algorithme de coordination des participants. Ce protocole est isolé par le motif de conception Strategy localisé dans la transaction. Ainsi, pour passer d'un protocole de validation en deux phases à un protocole de validation en une phase, il suffit de modifier le descripteur de l'architecture de la transaction pour qu'il utilise l'implantation de la nouvelle stratégie.

Le listing 7.4 donne un exemple d'adaptation statique du protocole au niveau du descripteur de l'architecture où la définition `TwoPhaseCommit` (ligne 13) du motif de conception Strategy est remplacée par la définition `OnePhaseCommit` (ligne 14). Dans le chapitre 11, nous illustrons la réalisation d'une telle adaptation de manière automatique à l'exécution.

```
1 <definition name="Prototype" extends="State,AutoExport(execute,Strategy),
2                                      AutoBind(Publish-Subscribe,subscribe,Notify),
3                                      AutoBind(Adapter,participant,Participant)">
4   <component name="Publish-Subscribe" definition="MessageBus"/>
5   <component name="Adapter"           definition="Adapter"/>
6   <component name="Strategy"          definition="Strategy"/>
7   <component name="State"             definition="State"/>
8   <binding client="adapter.state" server="state.state"/>
9 </definition>

11 <definition name="TransactionJTS" extends="Prototype,AutoInclude(Adapter,TransactionAdapterJTS),
12                                           AutoSharing(State,AtomicTransactionState)">
13   <!--<component name="Strategy" definition="Protocol2PC">-->
14   <component name="Strategy" definition="Protocol1PC"/>
15   <component name="Command1" definition="SynchronizationCommands"/>
16   <component name="Command2" definition="XAResourceCommands"/>
17 </definition>
```

LST. 7.4: Adaptation du protocole de validation.

7.6.2 Adaptation des participants d'une transaction

L'adaptation des participants d'une transaction peut revêtir deux aspects. L'adaptation du comportement d'un type de participant consiste à modifier par exemple, le composant Synchronization Commands présenté dans la figure 7.11 pour étendre la liste des commandes réalisables sur les participants de type `Synchronization`. Chaque commande étant isolée dans un composant primitif, il suffit de développer un nouveau composant représentant la commande à réaliser et d'intégrer ce dernier dans le composant Synchronization Commands.

Le second type d'adaptation réalisable au niveau des participants d'une transaction consiste à modifier le type de participant supportés par une transaction. Ce type d'adaptation requiert de changer l'implantation complète du composant Synchronization Commands pour le remplacer par la liste des commandes réalisables sur le nouveau type de composant. Dès lors, il suffit de modifier le descripteur de la transaction présenté dans le listing 7.4 pour remplacer la définition `SynchronizationCommands` (ligne 15) par une nouvelle définition.

7.6.3 Adaptation d'un standard transactionnel

L'adaptation de l'interface d'un standard de transaction consiste à remplacer l'implantation du motif de conception Adapter par une implantation différente.

Dans le listing 7.5 présente l'adaptation d'un service de transactions générique `Service` pour qu'il supporte les interfaces définies par la spécification *Java Transaction API*. Cette adaptation requiert une intervention au niveau des interfaces du service de transactions et au niveau des transactions. Pour ce faire, nous utilisons la définition `TransactionJTS` présentée dans le listing 7.4 en argument de la définition `Service`. Cette définition `Service` est ensuite complétée avec la définition `ServiceAdapterJTS` afin que le service fournisse les interfaces de programmation requises par le standard JTS.

```
1 <definition name="Service" arguments="prototype">
2   <component name="Adapter"   definition="AbstractServiceAdapter"/>
3   <component name="Singleton" definition="ThreadSingleton"/>
4   <component name="Factory"   definition="CacheFactory(${prototype})"/>
5   <binding client="Adapter.singleton" server="Singleton.singleton"/>
6   <binding client="Adapter.factory"   server="Factory.factory"/>
7   <binding client="Singleton.factory" server="Factory.factory"/>
8 </definition>

10 <definition name="ServiceJTS"
11         extends="Service(TransactionJTS),AutoInclude(Adapter,ServiceAdapterJTS)"/>
```

LST. 7.5: Adaptation d'un service de transactions de type JTS.

7.7 Travaux connexes

Cette section compare notre proposition aux travaux réalisés sur les canevas intergiciels et sur l'utilisation des aspects pour la modélisation des motifs de conception.

7.7.1 Canevas intergiciels

Les canevas intergiciels à base de composants visent la construction de systèmes plus configurables et administrables que les approches traditionnelles. Des exemples de tels intergiciels sont DREAM [Qué05], MEDOR [ACBD+04], OPENORB [BCA+01] ou encore JONATHAN [DHTS99]. Pour ce faire, ces intergiciels marient l'utilisation des principes de canevas, des notions de composants et des mécanismes de réflexion afin de faciliter la maintenance et l'évolution de l'architecture des systèmes cibles.

Cependant, la structuration de ces architectures est rarement adressée. En effet, si ces canevas supportent l'extension et la reconfiguration de leur architecture, peu d'entre eux guident l'évolution de l'architecture en proposant un paradigme particulier. Dans GOTM, nous proposons d'utiliser les motifs de conception comme paradigme de base pour l'évolution de l'architecture. Les motifs de conception proposent non seulement des solutions à un ensemble de problèmes identifiés mais ils sont également un moyen puissant pour comprendre, modéliser des architectures de très fine granularité. Nous montrons que l'utilisation des motifs de conception dans GOTM nous permet de définir une abstraction de l'architecture d'un service de transactions. Puis, en réalisant ces motifs de conception sous forme de composants de fine granularité, nous offrons une structure extensible et adaptable aux différentes dimensions de variabilité d'un service de transactions (protocoles de validation, standards transactionnels, etc.).

7.7.2 Aspects et motifs de conception

L'application des motifs de conception et des aspects a déjà fait l'objet de nombreuses études par le passé. L'objectif des études est d'automatiser l'application des motifs de conception dans les applications afin d'en faciliter la maintenance et l'évolution. Dans [HK02, DAAC05], les auteurs proposent d'introduire des motifs de conception dans une application existante en utilisant le paradigme d'aspect. L'utilisation du paradigme d'aspect permet d'introduire les aspects de façon transparente dans l'application et évite que la connaissance du motif ne soit perdue à l'exécution après application sur l'application. Dans UMLAUT [LSJ00, HJPP02], les auteurs s'attachent à décrire les motifs de conception à appliquer durant la conception de l'application puis ils définissent un tisseur dédié pour intégrer automatiquement les motifs de conception dans l'application.

Si ces approches facilitent la maintenance et l'évolution d'application à base d'objets, elle ne fournissent aucun support architectural pour cette évolution. Dans GOTM, nous nous appuyons sur le paradigme de programmation par composants pour isoler les entités caractéristiques d'un service de transactions. L'utilisation des motifs de conception nous permet d'identifier des motifs d'assemblage architecturaux des services de transactions. Ainsi, les motifs de conception que

nous employons ne se contentent pas uniquement de répondre aux problèmes d'assemblage que peut poser la définition d'un service de transactions mais ils facilitent également l'adaptation et l'évolution des services de transactions construits avec GOTM. Ainsi, les différentes personnalités de services de transactions réalisables avec GOTM (par exemple, JTS, OTS ou WS-AT) correspondent à des extensions maîtrisées des motifs de conception utilisés par GOTM.

7.8 Conclusion

Ce chapitre a présenté les principaux composants du canevas logiciel GOTM. Outre les principes traditionnels liés au développement des intergiciels, le canevas GOTM exploite les notions de motif de conception et de composants de granularité extrêmement fine afin d'accroître les possibilités d'extension et d'adaptation du canevas. Nous proposons une caractérisation complète du service de transactions en utilisant des motifs de conception reconnus. Chaque motif de conception représente ainsi un point de variation potentiel du service de transactions. Nous avons ensuite appliqué l'approche de conception par composants pour réaliser les motifs de conception. Cette réalisation permet de réifier au niveau de l'architecture leurs propriétés d'extensibilité afin de supporter les adaptations du service de transactions que nous avons pu identifier.

Le canevas logiciel GOTM correspond donc à une bibliothèque de composants techniques utilisables pour construire de nombreux services de transactions. Les composants fournis par GOTM sont développés avec FRACLET alors que les motifs de conception exploités par GOTM sont décrits à l'aide de notre langage de description et de vérification de motifs d'architecture. Un service de transactions concret est par conséquent obtenu par raffinement et spécialisation de cette architecture de base. Cette étape consiste à étendre les descriptions FRACTAL ADL du service de transactions pour décrire le protocole de validation, l'automate des états transactionnels, les différents types de participants et le standard transactionnel associé au service de transactions concret.

Cependant, la validité sémantique de ces assemblages ne peut être vérifiée au niveau des descriptions à cause de l'uniformisation des différents concepts sous forme de composants. Il est donc nécessaire d'utiliser un formalisme plus riche pour décrire et vérifier la sémantique des différentes caractéristiques d'un service de transactions. Le chapitre suivant détaille une proposition de modélisation des configurations des services de transactions en utilisant des modèles de haut niveau.

Le canevas logiciel GOTM est disponible librement à l'adresse suivante : `http://gotm.objectweb.org/`. Les travaux présentés dans ce chapitre ont donné lieu à plusieurs publications [RM04a, RM06b, RSAM06a, RSAM06b].

Program complexity grows until it exceeds the capability of the programmer who must maintain it.

Murphy's Law

Chapitre 8

Les modèles dédiés à la configuration des services de transactions

Sommaire

8.1 Introduction . . . 144

8.2 Motivations et objectifs . . . 145

8.2.1 Motivations de l'ingénierie dirigée par les modèles . . . 145

8.2.2 Objectifs de l'application à la configuration des canevas intergiciels . . . 145

8.3 Modélisation du standard . . . 146

8.3.1 Modèle d'assemblage du standard transactionnel . . . 147

8.3.2 Modélisation du standard transactionnel JTS . . . 148

8.3.3 Éléments d'implantation . . . 149

8.4 Modélisation des états . . . 149

8.4.1 Présentation du diagramme UML d'états . . . 151

8.4.2 Modélisation de la machine à états d'une transaction . . . 152

8.4.3 Éléments d'implantation . . . 153

8.5 Modélisation des participants . . . 153

8.5.1 Présentation des règles Événement—Condition—Action . . . 154

8.5.2 Modélisation des participants de type *Synchronization* . . . 154

8.5.3 Éléments d'implantation . . . 156

8.6 Modélisation du protocole de validation . . . 156

8.6.1 Présentation du diagramme UML de séquences . . . 157

8.6.2 Modélisation du protocole de validation en deux phases . . . 157

8.6.3 Éléments d'implantation . . . 159

8.7 Conclusion . . . 159

Ce chapitre présente notre contribution sur les aspects de configuration des services de transactions construits à l'aide du canevas GoTM*. En effet, la configuration des intergiciels basés sur des canevas à base de composants demeure une tâche délicate que bien souvent seul l'architecte du canevas peut maîtriser. Afin de fournir une abstraction plus intelligible de l'architecture de ces intergiciels et d'en simplifier la configuration, nous proposons d'utiliser plusieurs modèles de haut niveau manipulables par les intégrateurs de personnalités du canevas logiciel.*
Dans le cadre du canevas GoTM*, chaque modèle est adapté à une caractéristique du service de transactions. Chaque configuration exprimée par l'intégrateur en utilisant l'un de ces modèles est ensuite transformée vers un assemblage de composants de la bibliothèque* GoTM*. Cette transformation applique des règles de conversion des concepts spécifiques au modèle vers les descripteurs d'architecture imposés par le canevas logiciel* GoTM*. En particulier, nous utilisons les diagrammes UML d'états et de séquences pour représenter le modèle de transactions et le protocole de validation. L'utilisation d'un modèle à base de règles Événement—Condition—Action nous permet de décrire le comportement des participants d'une transaction. Enfin, à partir d'un méta-modèle dédié, nous sommes en mesure d'exprimer les fonctionnalités fournies et requises pour un standard transactionnel donné.*

8.1 Introduction

Les canevas logiciels à base de composants tels que DREAM [Qué05], PERSEUS [ACBD+04] ou OPENCOM [CBG+04] ont désormais suffisamment de maturité pour être utilisés dans des plates-formes applicatives telles que JORAM [QBFL04], JONAS [Exe04], ou OPENORB [BCA+01]. Ces intégrations consistent souvent à fournir une configuration figée du canevas logiciel particulièrement adaptée à la plate-forme applicative considérée. Cependant, l'aspect statique de cette configuration ne permet pas de bénéficier de toute la flexibilité offerte par l'utilisation des composants. Ce manque de flexibilité est bien souvent dû à la complexité des assemblages du canevas logiciel utilisé. Ces assemblages intègrent de nombreux aspects techniques spécifiques au canevas logiciel et qui sont peu intelligibles par les intégrateurs. Il est donc nécessaire d'utiliser un mécanisme de configuration plus riche que les définitions d'architecture pour que les plates-formes applicatives puissent bénéficier de la flexibilité des canevas logiciels à base de composants.

Dans le cadre de notre canevas logiciel GoTM, ce mécanisme de configuration prend la forme d'un ensemble de modèles de haut niveau indépendants qui adressent les différentes caractéristiques configurables d'un service de transactions. Il est ainsi possible de décrire le comportement du protocole de validation d'une transaction ou l'automate de ses états en utilisant un formalisme graphique plus intelligible pour l'intégrateur du canevas logiciel. Ce formalisme de haut niveau permet également de réaliser un certain nombre de vérifications sémantiques liées à la préoccupation technique du service de transactions considéré. Par exemple, il est possible de vérifier que l'automate d'états d'une transaction est valide. Des vérifications peuvent également être réalisées sur un ensemble de modèles de haut niveau pour vérifier par exemple que la modélisation d'un protocole de validation respecte bien les transitions autorisées par son automate d'états. Ces modèles de haut niveau sont ensuite transformés en assemblages de composants de la bibliothèque GoTM. Cette transformation convertit les concepts des modèles de haut niveau vers les abstractions architecturales du canevas GoTM par l'application d'un jeu de règles de transformation.

La suite de ce chapitre présente les quatre modèles de haut niveau que nous avons utilisés pour configurer un service de transactions en utilisant le canevas logiciel GoTM. Dans la section 8.2, nous introduisons les motivations d'une approche dirigée par les modèles et les objectifs quant à son application dans le canevas GoTM. La section 8.3 détaille l'adaptation du canevas GoTM à un standard transactionnel. Dans la section 8.4, nous présentons la configuration de

l'automate d'états d'une transaction. Puis, dans la section 8.5, nous présentons la configuration des participants d'une transaction. Enfin, la section 8.6 introduit la description du protocole de validation d'une transaction et la section 8.7 conclut.

8.2 Motivations et objectifs

Dans cette section, nous nous intéressons aux principes définis par l'ingénierie dirigée par les modèles afin d'observer quels peuvent être les apports d'une démarche MDE dans le cadre de la construction d'un canevas intergiciel hautement adaptable.

8.2.1 Motivations de l'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles vise à assurer la pérennité du système d'information face à l'évolution des technologies de mise en œuvre [MDBF06]. En effet, durant ces dernières années, le nombre de paradigmes de programmation utilisables pour la réalisation d'un logiciel déterminé n'a cessé de croître. Cette multiplication des modèles de programmation et la concurrence qu'ils se livrent rend de plus en plus complexe le choix de la technologie à appliquer pour implanter un système d'information particulier. De plus, lors de l'évolution de la plate-forme d'exécution du système, les caractéristiques du système d'information sont rarement capitalisées dans des éléments réutilisables et indépendants des technologies de mise en œuvre. De même, l'évolution du système d'information peut, dans certains cas, impacter une grosse partie de l'application existante et nécessiter de nombreuses modifications de la base de code. Ces limitations face aux évolutions possibles d'un système nuisent à assurer la pérennité des applications existantes en introduisant une dépendance forte vers les technologies de mise en œuvre sans pour autant capitaliser les connaissances liées au métier de l'application.

Pour remédier à ces limitations, l'ingénierie dirigée par les modèles propose de capitaliser les connaissances liées au métier dans des modèles de haut niveau indépendants des technologies de mise en œuvre. Puis, l'ingénierie dirigée par les modèles définit le processus de transformation comme un mécanisme permettant de convertir la connaissance contenue dans les modèles vers une application exécutable réalisée avec une technologie déterminée. Par conséquent, le choix de la technologie est découplé de la représentation du système d'information et son intégration *a posteriori* permet de résister à l'apparition, l'évolution ou la disparition des technologies de mise en œuvre. La figure 8.1 introduit une représentation de la démarche MDE. Dans cette démarche, le modèle indépendant de la plate-forme (*Platform-Independent Model* [PIM]) capitalise les connaissances du système d'information. L'application d'une opération de **transformation** convertit le PIM en un modèle spécifique à une plate-forme (*Platform-Specific Model* [PSM]). Un des principes de la démarche MDE est sa capacité à pouvoir convertir un PIM vers différents types de PSM. Chaque PSM intègre alors les caractéristiques d'une technologie de mise en œuvre particulière. Puis, par une opération de **génération**, le code métier de l'application est inféré à partir du PSM. Cette application implantée avec la technologie associée au PSM utilise alors une plate-forme logicielle adéquate comme support d'**exécution**.

8.2.2 Objectifs de l'application à la configuration des canevas intergiciels

Les canevas intergiciels correspondent quant à eux à des bibliothèques de fonctions élémentaires. Ces fonctions élémentaires peuvent être composées pour fournir une personnalité déterminée d'un intergiciel. La composition de ces fonctions repose généralement sur l'utilisation d'un langage de description d'architectures. Or, si le canevas intergiciel est en mesure de capitaliser le code des fonctions indépendamment des personnalités des intergiciels considérés, l'architecture des personnalités demeure une construction *ad-hoc* dont la maîtrise peut se révéler plus complexe qu'elle n'y paraît pour l'intégrateur d'intergiciel. Dès lors, nous souhaitons étudier l'apport de la démarche MDE dans le cadre de la capitalisation des éléments d'architecture relatif à l'intégration d'un canevas intergiciel. En particulier, nous souhaitons utiliser les modèles

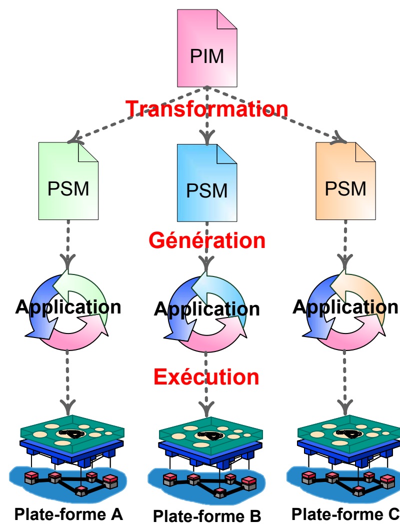


FIG. 8.1: L'ingénierie dirigée par les modèles.

pour configurer les assemblages des fonctions élémentaires fournies par notre canevas intergiciel GOTM.

Par conséquent, notre proposition consiste à utiliser différents modèles de haut niveau pour modéliser et capitaliser la complexité des fonctionnalités transactionnelles ; puis de transformer automatiquement ces modèles vers les artefacts architecturaux supportés par notre canevas GOTM. Après avoir séparé les différentes préoccupations d'un service de transactions, nous isolons chaque préoccupation dans un modèle qui est particulièrement approprié à sa modélisation. C'est ainsi que l'automate d'états d'une transaction peut être modélisé par un diagramme UML d'états. Le protocole de validation d'une transaction peut être représenté par un diagramme UML de séquences. Ensuite, le support des types de participants impliqués dans une transaction peut être modélisé par un jeu de règles Événement—Condition—Action. Enfin, la modélisation d'un standard transactionnel à l'aide d'un modèle dédié vise non seulement à décrire le support des interfaces de programmation imposés par le standard, mais également à assurer l'adéquation des préoccupations qu'il met en jeu. Nous utilisons ensuite différents mécanismes de transformation et de génération configurés par des règles propres à GOTM pour obtenir l'architecture concrète d'une personnalité de service de transactions.

Notre objectif est ainsi de capitaliser et d'isoler les préoccupations d'un service de transactions dans des modèles indépendants de la technologie utilisée par notre canevas GOTM. Ces modèles fournissent une sémantique riche et adaptée à la préoccupation qui peut être aisément interprétée par l'intégrateur du service de transactions. L'utilisation de ces modèles de haut niveau nous prémunit des évolutions possibles de la technologie de mise en œuvre et facilite l'intégration de nouvelles stratégies. La suite de ce chapitre détaille la définition et l'utilisation de modèles de haut niveau pour automatiser la construction des personnalités des services de transactions.

8.3 Modélisation du standard

Dans la mesure où les services de transactions ont fait l'objet de nombreux efforts de standardisation par les plus grands organismes internationaux de standardisation, il est nécessaire d'adresser la modélisation de ces standards pour conserver la compatibilité entre les services de transactions construits avec le canevas logiciel GOTM et les standards transactionnels.

8.3.1 Modèle d'assemblage du standard transactionnel

Au cours de l'une de nos précédentes expériences, nous avons défini un méta-modèle dédié à la génération des composants d'adaptation au standard transactionnel [RM04b]. Ce méta-modèle est composé d'une partie structurelle et d'une partie comportementale permettant respectivement de décrire la structure d'un jeu d'interfaces et de convertir les méthodes décrites par les standards vers les méthodes définies par notre canevas GOTM. Cependant, il s'avère qu'à l'utilisation ce modèle (1) ne couvre pas tous les aspects de la spécification d'un standard transactionnel et (2) propose une syntaxe aussi verbeuse que l'utilisation d'un langage de programmation pour décrire la conversion des interfaces. En effet, un standard transactionnel ne se limite pas à la simple définition d'un jeu d'interfaces de programmation qui lui sont spécifiques mais il définit également la sémantique que le service de transactions doit fournir. Pour conséquent, nous proposons deux éléments de réponse pour simplifier le support des standards transactionnels.

Tout d'abord, nous préconisons l'utilisation de notre modèle de programmation FRACLET pour exprimer la conversion des interfaces du standard transactionnel vers les interfaces du canevas GOTM. Cette description de la conversion des interfaces du standard transactionnel, nous permet de générer les composants implantant le motif de conception ADAPTER que nous avons présenté dans la section 7.5.1. Ensuite, nous définissons un méta-modèle dédié à la description de l'assemblage d'un service de transactions supportant un standard transactionnel particulier. Les configurations décrites en utilisant ce méta-modèle servent à assembler les composants fournis par le canevas GOTM pour réaliser une personnalité de service de transactions particulière.

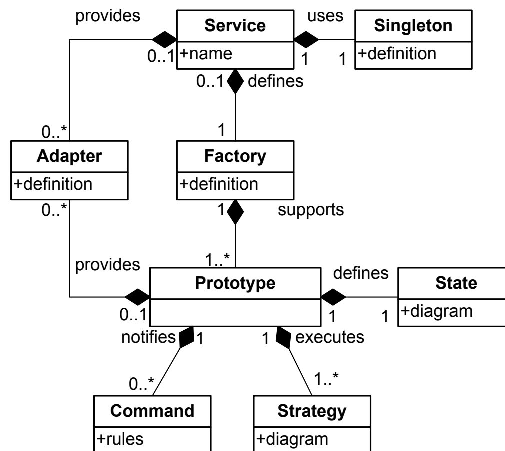


FIG. 8.2: Méta-modèle d'assemblage du standard transactionnel.

Pour décrire la configuration du standard transactionnel, nous avons recouru à la définition d'un méta-modèle dédié permettant d'associer les éléments caractéristiques des standards transactionnels aux composants fournis par le canevas logiciel GOTM. La figure 8.2 présente le méta-modèle d'assemblage du service de transactions que nous avons défini en nous appuyant sur les motifs de conception identifiés dans le chapitre 7. L'élément racine de ce méta-modèle est par conséquent l'élément Service. Un service peut éventuellement fournir plusieurs éléments Adapter[1] auxquels il est nécessaire d'associer une définition FRACTAL ADL. Le service spécifie également une politique de Singleton à utiliser pour gérer les instances manipulées. Ensuite, le service décrit un élément Factory dont l'attribut definition précise le type d'usine à utiliser. L'élément Factory peut être associé à un ou plusieurs éléments Prototype[2]. Chaque élément Prototype peut définir plusieurs éléments Adapter, un élément State, et plusieurs éléments Strategy et Command. La sémantique de l'élément Adapter est de fournir une façade pour les instances manipulées par le service de transactions. L'élément State est paramétré par un diagramme UML

[1] Une exemple de service de transactions supportant plusieurs adaptateurs est présenté dans le chapitre 10.
[2] Un exemple de service de transactions supportant plusieurs prototypes est présenté dans le chapitre 11.

d'états décrivant l'automate d'états de la transaction. Les éléments Strategy sont paramétrés par des diagramme de séquences UML décrivant les protocoles de validation ou d'abandon de la transaction. Enfin, les éléments Command sont paramétrés par des jeux de règles ECA décrivant la réaction des participants lors de l'occurrence de certains évéments émis par la transaction.

L'utilisation de ce modèle d'assemblage permet de configurer simplement les caractéristiques d'un service de transactions en décrivant le comportement à appliquer pour les différents motifs de conception identifiés dans le chapitre 7.

8.3.2 Modélisation du standard transactionnel JTS

Dans cette section, nous illustrons la description d'un service de transactions conforme au standard transactionnel JTS. Le listing 8.1 présente une configuration possible du service de transactions JTS. Cette configuration définit un seul adaptateur de service (`ServiceAdapterJTS`), ce qui signifie que le service ne supporte que le standard transactionnel JTS. La politique de singleton `ThreadSingleton` assure une isolation par *thread* des transactions. La politique `CacheFactory` associée à l'usine de transactions utilise une mécanisme de cache pour recycler les instances de transactions et accroître les performances du service de transactions. L'usine de transactions s'appuie sur un seul prototype de transactions. Ce patron fournit un adaptateur `TransactionJTS` précisant les interfaces fournies par une transaction JTS. Cette transaction est configurée par le diagramme UML d'états `AtomicTransactionState` (introduit dans la section 8.4) et le diagramme UML de séquences `Protocol2PC` (introduit dans la section 8.6). Les types de participants supportés par la transaction est décrit par les commandes définies par les règles ECA `SynchronizationRules` (présentées dans la section 8.5) et `XAResourceRules`.

```
1 <definition name="ServiceJTS">
2   <adapter definition="ServiceAdapterJTS"/>
3   <singleton definition="ThreadSingleton"/>
4   <factory definition="CacheFactory">
5     <prototype>
6       <adapter definition="TransactionAdapterJTS"/>
7       <state diagram="AtomicTransactionState"/>
8       <strategy diagram="Protocol2PC"/>
9       <command rules="SynchronizationRules"/>
10      <command rules="XAResourceRules"/>
11    </prototype>
12  </factory>
13 </service>
```

LST. 8.1: Description d'un service de transactions de type JTS.

Le listing 8.2 présente le descripteur FRACTAL ADL obtenu après transformation de la description du service de transactions. Les descriptions FRACTAL ADL ainsi générées utilisent non seulement les motifs génériques que nous avons définis dans la section 6.4.2 du chapitre 6, mais aussi les composants générés à partir des différents modèles de haut niveau que nous allons présenter dans ce chapitre. La description FRACTAL ADL `ServiceJTS` représente la partie statique du service de transactions (lignes 1–8). La description FRACTAL ADL `TransactionJTS` s'appuie sur la définition FRACTAL ADL `Prototype` (présentée dans le listing 7.4 du chapitre précédent) pour décrire la partie dynamique du service de transactions (lignes 10–15).

La figure 8.3 présente l'instance du service de transactions qui doit être construite à partir de la configuration exprimée dans le listing 8.1. La partie statique du service du service est représentée par le composant Service JTS alors que la partie dynamique correspond au composant Transaction JTS. Le composant Atomic Transaction State correspond au composant décrit dans la figure 8.7 et généré à partir du diagramme UML d'états. Le composant Protocol 2PC correspond au composant décrit dans la figure 8.13 et généré à partir du diagramme UML de séquences. Enfin, les composants Synchronization Commands et XAResource Commands ont une architecture similaire à l'architecture du composant décrit dans la figure 8.9 et sont obtenus après compilation des règles ECA associées. Ces composants générés à partir de nos modèles de haut niveau

```
1 <definition name="ServiceJTS" extends="AutoInclude(Adapter,ServiceAdapterJTS)"/>
2   <component name="Adapter"   definition="AbstractServiceAdapter"/>
3   <component name="Singleton" definition="ThreadSingleton"/>
4   <component name="Factory"   definition="CacheFactory(TransactionJTS)"/>
5   <binding client="Adapter.singleton" server="Singleton.singleton"/>
6   <binding client="Adapter.factory"   server="Factory.factory"/>
7   <binding client="Singleton.factory" server="Factory.factory"/>
8 </definition>

10 <definition name="TransactionJTS" extends="Prototype,AutoInclude(Adapter,TransactionAdapterJTS),
11                                            AutoSharing(State,AtomicTransactionState)">
12   <component name="Strategy" definition="Protocol2PC"/>
13   <component name="Command1" definition="SynchronizationCommands"/>
14   <component name="Command2" definition="XAResourceCommands"/>
15 </definition>
```

LST. 8.2: Description FRACTAL ADL d'un service de transactions de type JTS.

viennent ensuite s'insérer dans l'architecture abstraite du service de transactions. Les composants de cette architecture abstraite sont configurés à partir des attributs *definition* qui ont été définis dans la description du service de transactions comme nous l'avons illustré dans le listing 8.2.

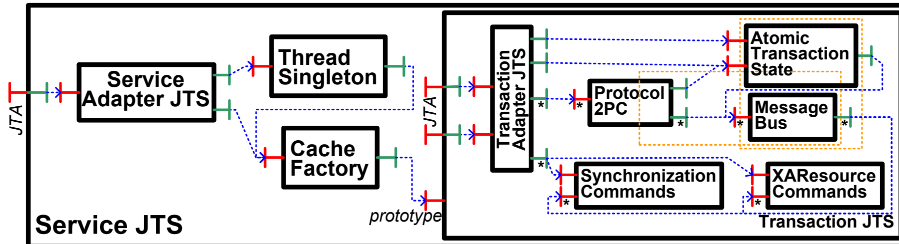


FIG. 8.3: Architecture du composant associé au standard JTS.

8.3.3 Éléments d'implantation

L'interprétation du modèle d'assemblage d'un service de transactions pour en construire une instance particulière correspond à la mise en place d'une chaîne de construction dont le rôle est de coordonner les différentes transformations pour aboutir à une description concrète du service à déployer. Cette chaîne de construction d'un service de transactions à partir du modèle d'assemblage est présentée dans la figure 8.4. Cette chaîne de production est initiée par l'**intégrateur** à partir des informations contenues dans le descripteur de standard. L'interprétation des différentes définitions FRACTAL ADL contenues dans ce descripteur permet de générer une architecture abstraite du service de transactions. Les diagrammes UML d'états et de séquences référencés par le descripteur sont ensuite transformés en descripteurs FRACTAL ADL comme présenté dans les sections 8.4 et 8.6. De façon similaire, l'implantation des gestionnaires de participants et la description FRACTAL ADL correspondante sont générés à partir des jeux de règles ECA (ECA+@) comme présenté dans la section 8.5. Ces différents descripteurs FRACTAL ADL s'appuient alors sur la bibliothèque de composants définie par le **développeur** et les motifs d'assemblage définis par l'**architecte** pour construire une implantation concrète du service de transactions.

8.4 Modélisation des états

L'automate d'états d'une transaction joue un rôle déterminant dans l'implantation d'un modèle de transactions déterminé. Cependant, la description de cet automate est rarement explicitée dans l'architecture d'un service de transactions. Nous cherchons par conséquent à modéliser

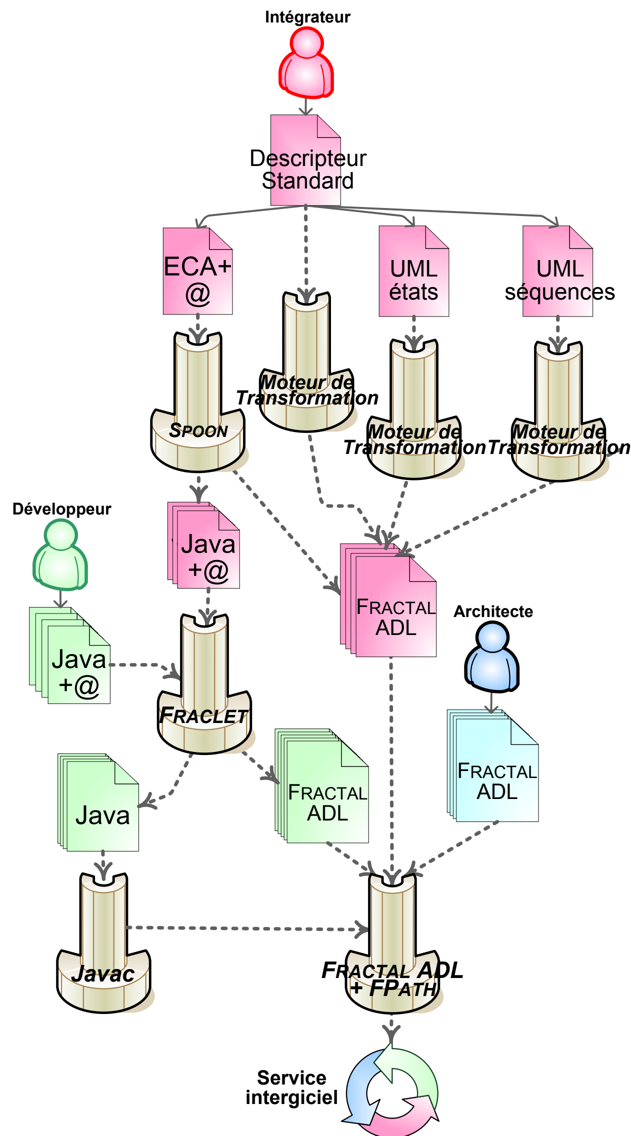


FIG. 8.4: Processus de construction d'un service de transactions.

de façon fiable l'automate d'une transaction et nous proposons d'utiliser le diagramme UML d'états pour réaliser cette modélisation. Cette description de haut niveau peut ensuite être transformée en une configuration du composant StateMachine fourni par la bibliothèque GoTM (présenté dans la section 7.5.5 du chapitre précédent).

8.4.1 Présentation du diagramme UML d'états

Le diagramme UML d'états présente une vue dynamique d'un système [OMG05b]. Le langage UML dispose donc d'une notation destinée à illustrer des événements et les états d'éléments tels que des transactions, des cas d'utilisation, des personnes, etc. Une machine à états UML présente les événements et les états intéressants d'un objet ainsi que son comportement face à un événement. Le cycle de vie d'un objet peut être ainsi représenté en montrant les événements auxquels il est soumis, ses transitions et les états par lesquels il passe entre chacun de ces événements.

Un événement est une occurrence d'un fait significatif ou remarquable. Un état est la condition d'un objet à un moment donné — il y demeure jusqu'à l'arrivée d'un nouvel événement. Une transition est une relation entre deux états qui indique que l'objet change d'état instantanément lorsqu'un événement se produit.

Règles de transformation vers le composant State. La transformation d'un diagramme UML d'états en un assemblage de composants est possible. La figure 8.5 présente l'ensemble des règles de transformation nécessaires à la configuration du composant State à partir d'un diagramme UML d'états.

- La règle (a) précise qu'un diagramme UML d'états est transformé en un composant composite étendant la définition StateMachine.
- La règle (b) définit que tout état du diagramme UML d'états doit être transformé en un composant de type State dont le nom correspond au nom de l'état.
- La règle (c) spécifie que le composant associé à l'état initial du diagramme UML doit être connecté à l'interface *initial* du composant State Manager.
- La règle (d) précise que l'attribut booléen **final** d'un composant état doit être positionné à **true** ; celui-ci modélise un état final dans le diagramme UML.
- La règle (e) définit une liaison entre deux composants états si et seulement si il existe une transition dans le diagramme UML d'états entre les deux états associés. La liaison définie porte le nom de l'événément déclencheur de la transition.

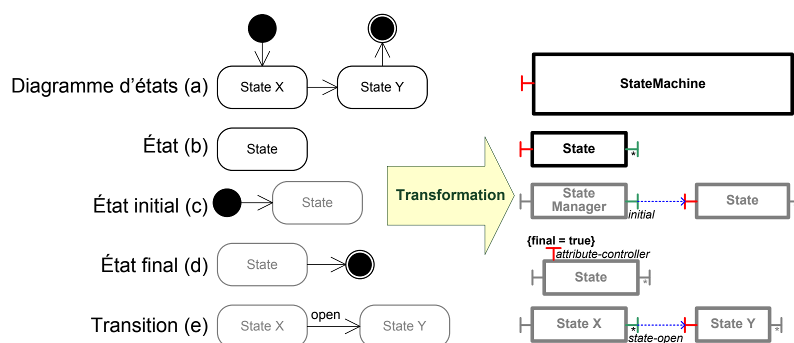


FIG. 8.5: Règles de transformation pour le diagramme UML d'états.

Les vérifications associées à ce type de diagramme permettent de s'assurer que l'automate dispose d'un seul état initial et d'au moins un état final. De plus, il est également possible de vérifier que l'automate ne dispose pas d'état puit qui traduirait un état incohérent de la transaction.

Dès lors il est possible de décrire différentes machines à états avec des diagrammes UML d'états et de configurer automatiquement des composants StateMachine réalisant ces machines

à états.

8.4.2 Modélisation de la machine à états d'une transaction

La figure 8.6 présente la machine à états d'une transaction plate en utilisant le diagramme UML d'états. À sa création, une transaction se situe dans l'état Inactive. La transaction devient Active lorsque l'événement *start* est déclenché. Toute transaction demeurant dans l'état Active ou MarkedAbort (atteint lorsque l'événement *mark-abort* est déclenché à partir de l'état Active) peut être suspendue et reprise en déclenchant les événements *suspend* et *resume*. Le protocole de validation en deux phases débute lorsque l'événement *prepare* est déclenché depuis l'état Active et que celle-ci passe dans l'état Preparing. Lorsque l'événément *abort* est déclenché depuis la plupart des états, la transaction passe alors dans l'état Aborting qui correspond à l'abandon de la transaction. Après réception de l'événement *vote*, le passage dans l'état Prepared décide de l'issue de la transaction. Si la transaction doit être validée, l'événement *commit* est déclenché et la transaction passe dans l'état Committing. Dans le cas contraire, l'événement *abort* est déclenché et la transaction passe dans l'état Aborting. À la réception de l'événement *acknowledge*, la transaction passe dans l'état Committed ou Aborted selon les cas.

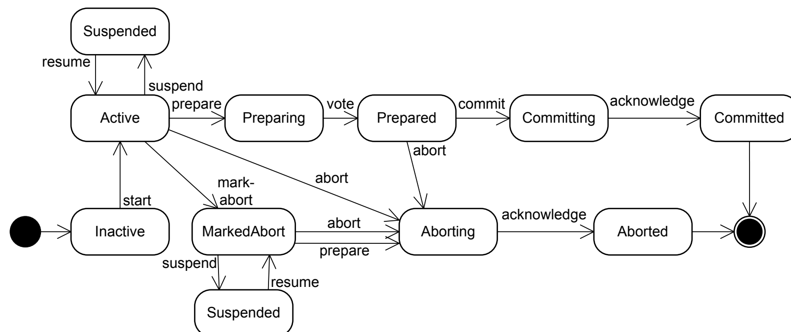


FIG. 8.6: Diagramme UML d'états d'une transaction plate.

En appliquant les règles de transformation présentées dans la figure 8.5, nous pouvons assembler un composant State s'appuyant sur la définition abstraite StateMachine et implantant la machine à états d'une transaction plate (comme détaillé dans le listing 7.1 du chapitre précédent). Ce composant, présenté dans la figure 8.7, permet d'assurer que l'évolution de l'état de la transaction à l'exécution respecte la spécification décrite dans le diagramme UML d'états. Ce composant propose également une représentation à l'exécution de l'automate d'états de la transaction. Notamment, ce type de représentation est nécessaire pour faciliter la reconfiguration dynamique d'une transaction.

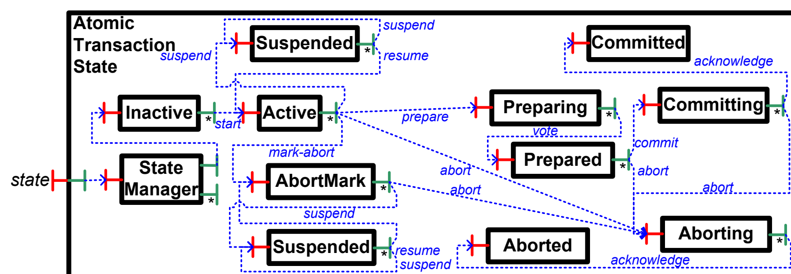


FIG. 8.7: Architecture du composant à états State d'une transaction plate.

En conclusion, l'utilisation du diagramme UML d'états permet de modéliser l'automate d'états d'un modèle de transactions déterminé et de générer automatiquement la description du

composant GoTM implantant la machine à états associée. Ce composant garantit à l'exécution le respect de la description de l'automate d'états.

8.4.3 Éléments d'implantation

L'identification des règles de conversion des concepts d'un diagramme UML d'états vers la construction du composant composite State nous permet d'automatiser le processus de transformation. La figure 8.8 introduit le processus de modélisation et de transformation du diagramme UML d'états en un assemblage de composants GoTM.

Dans un premier temps, l'intégrateur du canevas intergiciel utilise un *atelier de modélisation UML* quelconque pour modéliser le diagramme UML d'états du modèle de transactions. Ce diagramme UML d'états peut ensuite être exporté sous la forme d'un fichier XMI (*XML Metadata Interchange*) ou d'un modèle en mémoire. Les ateliers UML supportant ce type de fonctionnalités sont les outils ARGOUML [RR00], POSEIDON for UML[3], OBJECTEERING[4], ou encore RATIONAL ROSE MODELER[5]. En effet, notre proposition nous permet de nous appuyer sur les outils de modélisation existants et d'utiliser leurs fonctionnalités d'exportation de modèle pour obtenir une représentation manipulable du diagramme UML d'états.

Cette représentation du diagramme UML d'états peut ensuite être interprétée par un outil de transformation de modèle afin de générer le fichier de description FRACTAL ADL représentant l'assemblage du composant State. De nombreux outils de transformations supportent la génération de différents types d'artefacts (par exemple, descripteur, code) à partir de modèles en mémoire ou de fichiers XMI. Nous pouvons notamment citer les environnements KERMETA [MFJ05], MODTRANSF [Dum06] ou ATLAS TRANSFORMATION LANGUAGE (ATL) [JK05] comme des exemples d'outils pouvant nous permettre de convertir les concepts du diagramme UML d'états vers les constructions FRACTAL ADL de notre canevas intergiciel GoTM. Ces outils de transformation se reposent sur une représentation mémoire du modèle à transformer (par exemple, KERMETA, MODTRANSF) ou sur une représentation compatible avec le format XMI (par exemple, MODTRANSF, ATL) pour générer un fichier au format FRACTAL ADL décrivant le composant State.

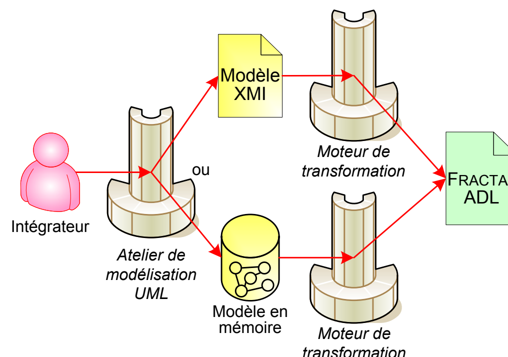


FIG. 8.8: Modélisation et transformation d'un diagramme UML d'états.

8.5 Modélisation des participants

Les participants impliqués dans l'exécution d'une transaction nécessitent d'être informés du déroulement de cette dernière afin de coordonner leurs états. Il existe de nombreux types de participants qui ne sont pas forcément notifiés de l'occurrence des mêmes événements d'une

[3] POSEIDON for UML : http://www.gentleware.com/
[4] OBJECTEERING : http://www.objecteering.com/
[5] RATIONAL ROSE MODELER : http://www-306.ibm.com/software/awdtools/developer/rose/modeler/

transaction. Nous proposons donc d'utiliser un modèle de règles Événement—Condition—Action (ECA) pour modéliser la réaction des types de participants en fonction des événements déclenchés par une transaction. Cet ensemble de règles ECA est utilisé pour configurer le composant Commands défini dans la bibliothèque GoTM.

8.5.1 Présentation des règles Événement—Condition—Action

Les règles ECA ont déjà été exploitées à plusieurs reprises dans les systèmes de bases de données actives [CC95] et dans les services de transactions [DHL90, ACV97] afin de fournir plus de flexibilité et de modularité à ces systèmes très fermés. Une règle ECA comporte trois parties : l'événement, la condition et l'action. Lorsque le système détecte l'occurrence d'un événement, il vérifie la condition de chaque règle associée à cet événement et exécute l'action définie par cette règle si la condition est vérifiée.

Pour ce faire, nous définissons notre propre modèle de programmation des règles ECA en étendant le langage Java avec un jeu d'annotations dédiées à la modélisation ECA. Le tableau 8.1 présente notre modèle de programmation des règles ECA :

- l'annotation @Rules marque une classe pour préciser qu'elle représente un jeu de règles ECA. L'attribut *name* permet de modifier le nom associé au jeu de règles,
- l'annotation @On marque une méthode comme réactive à l'occurrence d'un événement dont le type est spécifié par l'attribut *event*. La méthode annotée intègre l'action à réaliser et éventuellement une condition d'exécution spécifiée par l'instruction **`if`**,
- l'annotation @Global permet de partager une donnée entre plusieurs règles ECA. Par conséquent, un jeu de règles ECA est modélisé par une classe comportant au moins une méthode marquée avec l'annotation @On et éventuellement un ou plusieurs attributs marqués avec l'annotation @Global.

Annotation	Élément	Paramètre	Description	Contingence	Valeur par défaut
@Rules	`classe`	*name*	nom du jeu de règles	optionnel	signature de la classe
@On	`méthode`	*event*	type de l'événement déclencheur	optionnel	nom de la méthode
@Global	`champ`	-	-	-	-

TAB. 8.1: Jeu d'annotations pour le modèle de programmation de FRACTAL.

Nous utilisons ici les annotations Java 5 afin de pouvoir générer plusieurs artefacts de type identique à partir d'un même fichier de description des règles ECA. La définition d'un jeu d'annotations Java 5 nous permet d'utiliser notre outil de développement (par exemple, Eclipse[6]) favori afin de bénéficier de tous les avantages qui y sont associés (par exemple, compilation incrémentale, complétion, coloration syntaxique, aide contextuelle, assistants). Ainsi, la prise en charge de notre langage ECA ne nécessite pas de réaliser un nouvel analyseur syntaxique et un interpréteur pour être opérationnel.

8.5.2 Modélisation des participants de type *Synchronization*

Les participants de type *Synchronization* sont des entités impliquées dans l'exécution d'une transaction mais qui ne participent pas explicitement au protocole de validation. En effet, ces participants se contentent d'être notifiés du début et de la fin de la validation d'une transaction sans être consultés lors de la phase de vote du protocole de validation. Comme illustré dans le listing 8.3, la modélisation de ce type de participant en utilisant des règles ECA consiste donc à détecter l'occurrence de l'événement `prepare` pour détecter le début de la validation d'une transaction et l'événement `acknowledge` pour détecter la fin de la validation. L'issue de la transaction est déterminée en utilisant l'élément global `Status`.

[6]Le projet Eclipse : `http://www.eclipse.org/`

```
1 @Rules(name="SynchronizationCommands")
2 public class Synchronizations {
3   @Global Status status;

5   @On(event="prepare")
6   void prepare(Synchronization synchronization) {
7     synchronization.beforeCompletion();
8   }
9   @On(event="acknowledge")
10  void ack(Synchronization synchronization) {
11    synchronization.afterCompletion(status.getStatus());
12 } }
```

LST. 8.3: Modélisation ECA des participants de type *Synchronization*.

Règles de transformation vers le composant Synchronization Commands. Les règles de transformation vers l'implantation des commandes sont définies de la manière suivante :

– une classe marquée par l'annotation @Rules est transformée en un assemblage de composants réalisant le système ECA,
– toute méthode marquée avec l'annotation @On est transformée en un composant dont le nom correspond au nom de la méthode annotée (par exemple, les lignes 5–8 implantent le composant `Prepare`),
– le composant généré fournit une interface de type `Command` dont le nom correspond à la valeur de l'attribut *event* spécifié au niveau de l'annotation @On (par exemple, la ligne 5 déclare une interface fournie nommée `prepare`),
– le contenu de la méthode annotée est utilisé pour implanter le comportemement de la méthode `execute()` définie par l'interface `Command` (par exemple, ligne 7),
– tout attribut de classe marqué avec l'annotation @Global est transformé en une interface requise pour chaque composant généré à partir du jeu de règles (par exemple, ligne 3).

La réalisation des composants réifiant les règles ECA utilise notre modèle de programmation FRACLET (présenté dans le chapitre 5). Grâce à FRACLET, le code technique des composants et les descripteurs d'architecture sont automatiquement générés.

Les règles de transformation de la modélisation ECA vers la description FRACTAL ADL Synchronization Commands (présentée dans le listing 8.4) sont définies de la manière suivante :

– toute classe marquée avec l'annotation @Rules est transformé en un composant composite dont le nom correspond au nom du jeu de règles (ligne 1),
– les méthodes annotées correspondent à la déclaration de composants composites abstraits si une annotation @Global est définie pour le jeu de règles ECA (lignes 3–4), ou de composants primitifs si aucune annotation @Global n'est utilisée,
– le composant composite exporte toutes les commandes en utilisant la définition `AutoExport` (présentée dans le chapitre 6) (ligne 2).
– chaque attribut marqué par l'annotation @Global correspond au partage d'un composant dont le type dépend du type de l'attribut marqué, La déclaration du partage de ce composant est réalisée via l'utilisation de la définition `AutoSharing` (présentée dans le chapitre 6) (ligne 2).

```
1 <definition name="SynchronizationCommands"
2         extends="AutoExport(command,Command),AutoSharing(status,AtomicTransactionState)">
3   <component name="prepare" definition="Prepare"/>
4   <component name="ack"     definition="AckComposite"/>
5 </definition>
```

LST. 8.4: Description FRACTAL ADL du composant SynchronizationCommands.

L'application des règles de transformation à la modélisation ECA nous permet de configurer le composant Synchronization Commands pour l'interaction avec les participants de type *Synchronization*. L'architecture obtenue est présentée dans la figure 8.9.

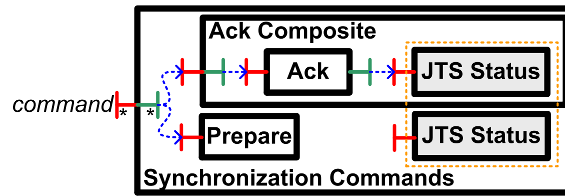


FIG. 8.9: Architecture du composant Synchronization Commands.

8.5.3 Éléments d'implantation

Pour réaliser la transformation de notre modèle de programmation des règles ECA vers l'assemblage de composants réalisant le motif de conception COMMAND, nous proposons d'utiliser SPOON. SPOON est un compilateur Java 5 ouvert construit au dessus de Javac [Paw05]. SPOON a déjà été utilisé pour réaliser une version FRACLET reposant sur les annotations Java 5. SPOON représente l'arbre abstrait de Java sous la forme d'un méta-modèle modifiable. SPOON utilise ce méta-modèle pour analyser les programmes Java 5. Le méta-modèle est ensuite modifié via l'utilisation de visiteurs appelés *processeurs*. Le parcours du méta-modèle est dirigé par les annotations contrairement à XDOCLET. Ce type de parcours nous permet de générer plusieurs composants à partir d'un même fichier source contenant le jeu de règles ECA.

En tirant avantage des fonctionnalités Java 5, SPOON fournit un canevas pour la définition de patrons de code en Java (en anglais, *template*). En spécifiant ces patrons de code en Java, le développeur peut utiliser son outil de développement favori afin de bénéficier de tous les avantages qui y sont associés (par exemple, compilation incrémentale, complétion, coloration syntaxique, aide contextuelle, assistants).

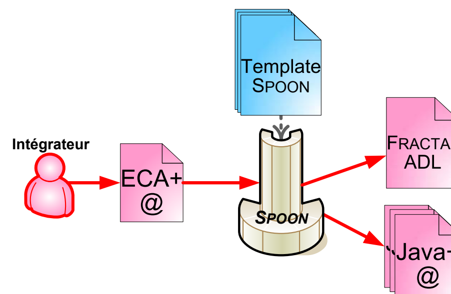


FIG. 8.10: Transformation des règles en ECA en composants FRACTAL.

Dès lors, la réalisation d'un patron de génération SPOON relatif à nos règles ECA consiste à développer différents fichiers *template* SPOON pour décrire les transformations à réaliser comme illustré dans la figure 8.10. Ces transformations consomment les annotations @On pour générer les composants réalisant chaque règle. Lors de la génération de ces composants, une interface requise est déclarée si une annotation @Global est détectée dans le jeu de règle ECA. Un second patron de génération SPOON permet de consommer les annotations @Rules pour générer le descripteur d'assemblage relatif au jeu de règles ECA. Un composant est déclaré pour chaque annotation @On présente dans la classe de règles. Ce composant est de type composite si au moins une annotation @Global est utilisée par la classe de règles ou primitif dans le cas inverse.

8.6 Modélisation du protocole de validation

Le protocole de validation d'une transaction correspond à un algorithme de coordination exécuté entre la transaction (généralement identifiée comme coordinateur) et les participants impliqués dans cette transaction. De nombreux protocoles de validation ont été définis ces dernières

années afin d'améliorer la tolérance aux fautes de cet algorithme ou de prendre en compte de nouveaux contextes d'exécution. Nous proposons ici d'utiliser le diagramme UML de séquences pour modéliser le déroulement de l'algorithme de coordination et générer à partir de celui-ci les assemblages de composants de la bibliothèque GOTM qui permettent d'implanter cet algorithme.

8.6.1 Présentation du diagramme UML de séquences

Un diagramme UML de séquences permet de décrire les interactions entre différentes entités et/ou acteurs [OMG05b]. Dans la figure 8.11, le temps est représenté comme s'écoulant du haut vers le bas le long des "lignes de vie" des entités. Des flèches représentent les messages qui transitent d'une entité vers l'autre. Le nom du message apparaît au dessus de chaque flèche. Si l'extrémité de la flèche est pleine, il s'agit d'une invocation. Une invocation peut être réflexive lorsque celle-ci est réalisée sur l'entité invocante. Si l'extrémité de la flèche est creuse, le message est alors synchone. Un retour de message doit être associé à un message synchrone. Enfin, si l'extrémité de la flèche est asymétrique, il s'agit d'un message asynchrone. Dans ce cas, aucun retour de message n'est nécessaire.

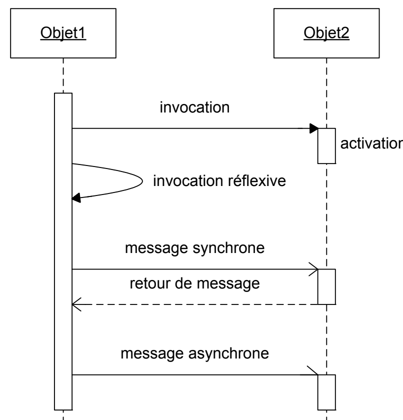


FIG. 8.11: Diagramme de séquences UML.

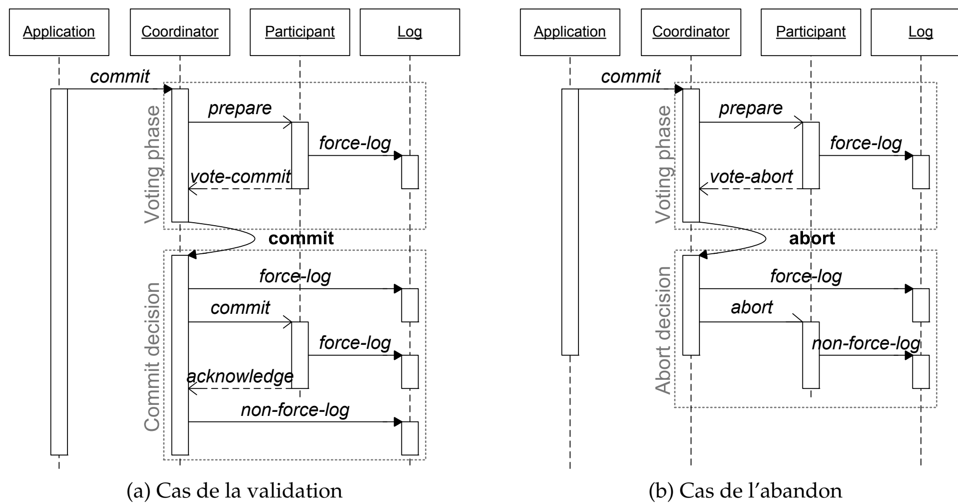
Le diagramme de séquences représente la succession chronologique des opérations réalisées par un acteur : saisir une donnée, consulter une donnée, lancer un traitement ; il indique les objets que l'acteur va manipuler, et les opérations qui font passer d'un objet à l'autre. On peut représenter les mêmes opérations par un diagramme de collaboration. Il s'agit d'un graphe dont les nœuds sont des objets et les arcs (numérotés selon la chronologie) les échanges entre objets. Les diagrammes de séquences et de collaboration sont deux vues différentes, mais logiquement équivalentes (on peut construire l'une à partir de l'autre), d'une même chronologie.

8.6.2 Modélisation du protocole de validation en deux phases

Les protocoles de validation en deux phases comportent deux étapes comme illustré dans la figure 8.12. Durant la phase de vote, le coordinateur envoie un message synchrone *prepare* à tous les participants. Durant la phase de décision, le coordinateur collecte les réponses des participants, et il décide de l'issue de la transaction. Cette issue peut être la validation (*commit* dans la figure 8.12a) si tous les participants ont voté *oui*, ou l'abandon (*abort* dans la figure 8.12b) si au moins l'un des participants a voté *non*. Lorsque l'issue est déterminée par le coordinateur, celui-ci envoie un message à tous les participants contenant l'issue qu'il a choisie. Lorsque les participants reçoivent cette décision, ils confirment ou infirment leurs modifications avant d'envoyer un accusé de réception (*acknowledge*) au coordinateur. Le protocole de validation se termine quand le coordinateur reçoit tous les accusés de réception des participants qui avaient voté *oui*

durant la phase de décision. Pour le protocole 2PC classique, le coordinateur force l'écriture de la décision prise quant à l'issue de la transaction (voir figure 8.12a). Il utilise une écriture non forcée pour mémoriser la fin du protocole. Les participants forcent l'écriture de leur vote ainsi que la décision qu'ils reçoivent du coordinateur. Ces opérations d'écriture sont réalisées avant l'envoi des messages correspondants.

Dans le cas du protocole 2PC-PA, la phase d'abandon de la transaction est différente de la phase de validation. Le protocole 2PC-PA réduit le coût associé aux transactions présentant un taux élevé d'abandon. Ainsi, lorsque le coordinateur décide d'abandonner la transaction, il élimine toutes les informations relatives à la transaction et il envoie un message *abort* à tous les participants sans mémoriser la décision (voir figure 8.12b). Les participants utilisent une écriture non forcée pour mémoriser le message *abort* et ne sont pas tenus d'envoyer un message d'accusé de réception au coordinateur. Toute information manquante dans le journal est donc interprêtée comme un abandon de la transaction. Le comportement de la validation d'une transaction utilisant le protocole 2PC-PA est en tout point identique à celui des transactions utilisant le protocole 2PC (voir figure 8.12a).



(a) Cas de la validation (b) Cas de l'abandon

FIG. 8.12: Diagramme de séquences UML du protocole 2PC-PA.

Les vérifications liées à ce modèle assurent que l'algorithme de coordination décrit par la séquence est valide.

La modélisation complète du protocole de validation 2PC et de deux de ses optimisations est présentée dans le chapitre 11.

Règles de transformation vers le composant Commit Protocol. Nous souhaitons pouvoir configurer l'implantation des protocoles de type 2PC en utilisant le diagramme de séquences UML. Pour cela, nous proposons un ensemble de règles de transformation d'un diagramme de séquences UML modélisant un protocole de type 2PC en un assemblage de composants réalisant ce protocole.

– Le diagramme de séquences UML est transformé en un composant Commit Protocol. Ce composant inclut un composant Message Bus qui propose deux modes d'émission des messages : le mode synchrone (interface *sync*) et le mode asynchrone (interface *async*).
– Chaque phase du protocole de l'entité Coordinator est isolée dans un composant (composants Vote, Commit et Abort). Ces composants présentent la même structure et peuvent être réutilisés. En effet, chacun d'eux dispose d'une interface cliente pour émettre un message et deux interfaces de log pour mémoriser les étapes du protocole avant et après l'émission du message. Le composant Vote dispose de deux interfaces clientes supplémentaires car le

résultat de la phase de vote doit déterminer la suite de l'exécution du protocole : la validation ou l'abandon.

– L'entité Participant est représentée par un composant Participants chargé de propager et de cumuler les messages envoyés aux participants. Ce composant présente une interface cliente collection pour mémoriser l'évolution du protocole du côté des participants. Les interfaces serveur du composant Participants sont connectées à l'interface cliente collection du composant Message Bus.
– Les messages échangés entre le coordinateur et les participants sont transformés en un ensemble de liaisons entre les composants Vote, Commit et Abort et le composant Message Bus. L'interface serveur du composant Message Bus utilisée pour la liaison dépend de la nature du message échangé entre le coordinateur et les participants, c.-à-d. synchrone ou asynchrone.
– L'entité Log est réalisée par le composant Log (présenté dans la section 4.3.2 du chapitre 4). Le libellé des interactions entre les entités Coordinator, Participant et Log détermine le nom de l'interface serveur du composant Log à utiliser lors de la liaison.

Tous ces composants sont fournis par le canevas logiciel GOTM et sont présentés dans la figure 8.13. L'objectif de la modélisation n'est pas de pouvoir synthétiser ces composants mais de décrire l'assemblage du composant Commit Protocol.

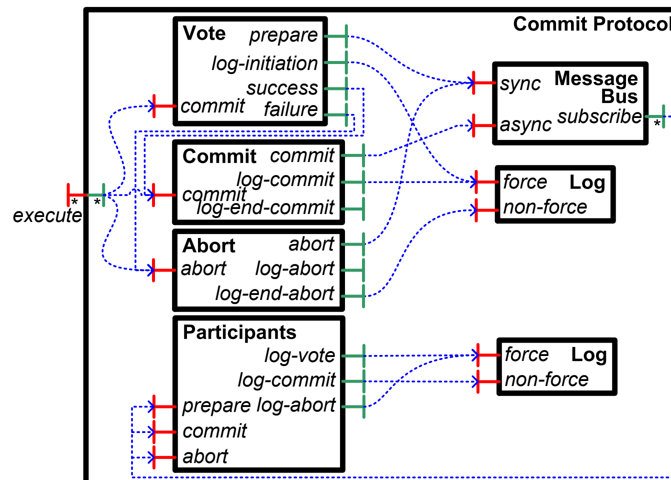


FIG. 8.13: Architecture du composant 2PC-PA.

8.6.3 Éléments d'implantation

La transformation du diagramme de séquences UML vers un assemblage de composants fournis par notre canevas GOTM est réalisable par l'application d'un processus semblable à celui utilisé pour la transformation des diagrammes UML d'états. Ce processus, présenté dans la section 8.4.3, permet d'utiliser les outils de modélisation existants pour décrire le diagramme de séquences du protocole de validation en deux phases. Puis, le fichier XMI exporté ou le modèle en mémoire de l'outil de modélisation peut être interprété par un outil de transformation de modèles pour produire le fichier de description FRACTAL ADL correspondant à l'assemblage du composant Commit Protocol.

8.7 Conclusion

Ce chapitre a présenté un ensemble de modèles de haut niveau dédiés à la configuration des services de transactions construits avec le canevas logiciel GOTM. En effet, les canevas intergi-

ciels actuels présentent le défaut de ne pas fournir de formalisme de configuration adapté au métier considéré.

Notre proposition consiste à utiliser différents modèles de haut niveau existants pour exprimer la configuration de différents aspects d'un service de transactions. Ainsi, nous proposons d'utiliser le diagramme UML d'états pour modéliser les différents états d'une transaction. Nous utilisons également le diagramme de séquences UML pour représenter le protocole de validation d'une transaction. Ensuite, nous avons recours au formalisme des règles ECA pour représenter l'intégration et le rôle des participants d'une transaction. Enfin, nous avons défini un méta-modèle dédié à la description d'un standard transactionnel particulier selon les motifs de conception de GOTM. L'utilisation de ces modèles de haut niveau nous permet de capitaliser les différents modèles de transactions de façon intelligible et indépendante des technologies d'implantation.

Si les travaux présentés dans ce chapitre ont été validés par plusieurs publications [RM04b, RSAM06a, RSAM06b], nous ne disposons pas encore à l'heure actuelle de l'outillage complet permettant d'automatiser le passage des modèles de haut niveau aux configurations de composants du canevas logiciel GOTM. Les perspectives de ces travaux adressent la vérification des descriptions réalisées avec les modèles de haut niveau et l'étude des dépendances qui peuvent exister entre les différents modèles. Par exemple, nous observons que les éléments du diagramme de séquences UML sont conditionnés par la description réalisée avec le diagramme UML d'états et il en va de même pour les règles ECA. Il est donc nécessaire à l'avenir d'étudier les relations qui unissent ces différents modèles de haut niveau.

Organisation de la troisième partie

La troisième partie du document correspond aux expériences que nous avons réalisées pour valider notre démarche de construction. Le chapitre 9 présente l'abstraction de la démarcation transactionnelle dans les plates-formes à base de composants. Cette abstraction, nommée **JOTDF**, permet d'intégrer les six politiques de démarcation généralement définies par les spécifications des modèles de composants dans différentes plates-formes à base de composants. Le chapitre 10 présente la composition de plusieurs standards transactionnels pour assurer l'interopérabilité transactionnelle de plusieurs applications patrimoniales hétérogènes. Cette composition est réalisée par le service de transactions **ATS** qui implante plusieurs standards transactionnels simultanément. Enfin, le chapitre 11 présente l'adaptation dynamique du protocole de validation en deux phases d'un service de transactions. Cette adaptabilité est supportée par le service de transactions **CATE** capable de reconfigurer le protocole de validation en observant le contexte d'exécution.

Troisième partie

Expérimentations et validation

Real programmers don't comment their code. If it was hard to write, it should be hard to understand.

Philington's Second Law

Chapitre 9

OTDF : l'intégration du support transactionnel dans les plates-formes applicatives

Sommaire

9.1 Introduction . . . 164
9.2 Présentation de la démarcation transactionnelle . . . 165
9.2.1 Politiques de démarcation . . . 165
9.2.2 Domaines transactionnels . . . 166
9.3 Défis soulevés par l'étude de la démarcation de transactions . . . 166
9.3.1 Abstraction de la démarcation de transactions . . . 167
9.3.2 Abstraction du service de transactions . . . 167
9.3.3 Organisation des politiques de démarcation . . . 168
9.3.4 Intégration de nouvelles politiques de démarcation . . . 168
9.4 OTDF : le canevas de démarcation transactionnelle . . . 168
9.4.1 Présentation du canevas OTDF . . . 168
9.4.2 Abstraction du service de transactions . . . 169
9.4.3 Abstraction de la démarcation transactionnelle . . . 170
9.4.4 Intégration de nouvelles politiques . . . 172
9.5 JOTDF : l'implantation d'OTDF en Java . . . 173
9.5.1 Adaptateur de service de transactions . . . 173
9.5.2 Domaine transactionnel . . . 173
9.5.3 Politique transactionnelle . . . 173
9.5.4 Architecture d'un jeu de politiques transactionnelles . . . 175
9.5.5 Utilisation des politiques par les plates-formes applicatives . . . 175
9.6 Expérimentations . . . 175
9.6.1 Contexte et scénario . . . 177
9.6.2 Évaluation mémoire . . . 177
9.6.3 Évaluation de performances . . . 178
9.7 Conclusion . . . 178

LES INTERGICIELS À BASE DE COMPOSANTS deviennent le substrat privilégié pour l'exécution d'applications distribuées dans des environnements de plus en plus ouverts et hétérogènes. D'un point de vue technique, ces intergiciels définissent la notion de conteneur pour prendre en charge les propriétés non-fonctionnelles d'une application (par exemple, transactions, sécurité). En particulier, la fonctionnalité transactionnelle est intégrée dans ces conteneurs via un ensemble de politiques de démarcation. Le rôle de ces politiques est d'automatiser le contrôle du contexte transactionnel dans lequel l'application doit s'exécuter. Cependant, ces politiques sont souvent très liées à un service de transactions particulier et elles sont rarement isolées du reste du conteneur.
Dans ce chapitre, nous proposons Open Transaction Demarcation Framework *(OTDF) : un canevas à trois niveaux pour construire des politiques de démarcation indépendantes des services de transactions et réutilisables par de nombreuses plates-formes applicatives. Nous montrons que notre implantation de ce canevas n'introduit pas de surcoût à l'exécution comparé aux implantations existantes de cette fonctionnalité. Notre approche propose une vision très fine de la démarcation transactionnelle structurée que nous implantons en utilisant des composants développés avec* FRACLET *et s'appuyant sur notre canevas* GOTM.

9.1 Introduction

L'introduction des intergiciels à composants a permis de mettre en évidence la notion de propriété non-fonctionnelle. C'est ainsi que la plupart des intergiciels à composants tels que les *Enterprise Java Beans* (EJB) [DK05], le *CORBA Component Model* (CCM) [OMG02] ou *Service Component Architecture* (SCA) [IBM05] ont défini dans leurs spécifications respectives un ensemble de propriétés non-fonctionnelles. Parmi celles-ci, il est possible de retenir les propriétés de sécurité et de transaction. Ce chapitre s'attache donc à étudier l'intégration d'une de ces propriétés à savoir la démarcation transactionnelle dans les plates-formes à composants. Il s'agit de définir un canevas logiciel facilitant non seulement cette intégration mais offrant également suffisamment de flexibilité pour autoriser un certain nombre de variations qui n'étaient pas forcément offertes jusque là par les plates-formes traditionnelles.

La contribution principale de ce chapitre est de proposer *Open Transaction Demarcation Framework* (OTDF) : un canevas logiciel à base de composants traitant de la construction des politiques de démarcation. Ce canevas est indépendant du service de transactions sous-jacent utilisé et de la plate-forme à composants dans laquelle il peut être intégré. Il est donc possible de créer différentes instances de politiques de démarcation transactionnelle à partir de ce canevas afin de les utiliser dans des plates-formes du type CCM, EJB ou SCA. Ces politiques correspondent à une vision fine de la fonction de démarcation transactionnelle dans une plate-forme applicative et nous utilisons notre modèle de programmation FRACLET (présenté dans le chapitre 5) pour les réaliser. De plus, l'architecture des politiques que nous définissons reposent sur les principes et les abstractions de notre canevas GOTM. En particulier, nous montrons que l'application du motif de conception COMMAND offre une structure extensible pour la définition de nouvelles politiques de démarcation sans introduire de surcoût à l'exécution.

Après avoir introduit la démarcation transactionelle dans la section 9.2, nous étudions les différents défis posés par cette problématique dans la section 9.3. Ensuite, le canevas logiciel à base de composants pour la construction de propriétés transactionnelles est détaillé dans la section 9.4 et son implantation est présentée dans la section 9.5. La section 9.6 présente les expérimentations que nous avons réalisées pour valider le canevas OTDF et la section 9.7 conclut ce chapitre.

9.2 Présentation de la démarcation transactionnelle

Cette section présente les différentes politiques de démarcation définies par les plates-formes applicatives existantes et les domaines transactionnels qui leur sont associées.

9.2.1 Politiques de démarcation

La démarcation transactionnelle constitue un moyen de garantir le contexte transactionnel d'une invocation. Plus simplement, la notion de politique transactionnelle permet d'évaluer si l'invocation d'une méthode doit être réalisée dans le cadre d'une transaction active ou pas. Les standards d'intergiciels à composants tels que CCM, EJB ou SCA définissent un ensemble de six politiques de démarcation afin de couvrir la plupart des situations pouvant se produire. Ces politiques sont décrites dans le tableau 9.1.

Politique de démarcation	Transaction cliente	Transaction conteneur
Supports	-	-
	Transaction 1	Transaction 1
Never	-	-
	Transaction 1	RAISES(**NEVER**)
Mandatory	-	RAISES(**MANDATORY**)
	Transaction 1	Transaction 1
Required	-	Transaction 2
	Transaction 1	Transaction 1
Not Supported	-	-
	Transaction 1	-
Requires New	-	Transaction 2
	Transaction 1	Transaction 2

TAB. 9.1: Politiques de démarcation définie par CCM et EJB

La description de ces politiques est la suivante :

Supports. La politique Supports est utilisée quand une opération de composant est en mesure de prendre en charge des transactions. Si une transaction a été activée par le client, elle sera alors propagée jusqu'au conteneur. Si aucune transaction n'est activée par le client, la méthode sera en mesure de s'exécuter sans comportement transactionnel.

Never. La politique Never impose que l'opération considérée ne doit pas être invoquée dans le contexte d'une transaction. Si cette condition n'est pas vérifiée, une exception — spécifiant que la clause *Never* n'est pas respectée — est déclenchée. Dans le cas contraire, l'opération peut être invoquée sur le composant.

Mandatory. La politique Mandatory impose que l'opération considérée doit être invoquée dans le contexte d'une transaction. Si cette condition n'est pas vérifiée, une exception — spécifiant que la clause *Mandatory* n'est pas respectée — est déclenchée. Dans le cas contraire, l'opération peut être invoquée sur le composant.

Required. La politique Required est utilisée lorsqu'une méthode requiert une exécution dans le contexte d'une transaction. Si une transaction a déjà été activée par le client, la politique propage alors la transaction à la méthode. Si ce n'est pas le cas, la politique active alors automatiquement une nouvelle transaction qui sera validée une fois l'invocation de la méthode réalisée.

Not Supported. La politique Not Supported assure que l'exécution d'une méthode sera réalisée en dehors de tout contexte transactionnel. Ainsi, si une transaction est déjà activée, la politique de démarcation suspendra la transaction le temps de l'invocation de la méthode puis procédera à sa réactivation.

Requires New. Finalement, la politique Requires New est requise lorsqu'une méthode souhaite s'exécuter dans une transaction locale. Si une transaction a été préalablement activée, la politique commence par suspendre celle-ci avant d'activer une nouvelle transaction. La transaction activée sera validée après exécution de la méthode puis la transaction suspendue sera réactivée.

9.2.2 Domaines transactionnels

Après observation du comportement des politiques de démarcation, il est possible de définir une organisation plus structurée de celles-ci basée sur le type d'interaction que peut avoir une politique avec le service de transactions sous-jacent. Ainsi, les politiques Never et Mandatory se contentent uniquement de consulter l'état de la transaction courante pour effectuer leur traitement. Les politiques Required et Requires New utilisent le service de transactions pour activer de nouvelles transactions. Finalement, les politiques Not Supported et Requires New peuvent recourir aux fonctionnalités d'interruption et de reprise d'une transaction dans certains cas. À partir de cette analyse, il est possible de répartir les six politiques de démarcation précédemment présentées entre trois domaines tel que cela est illustré dans la figure 9.1. La politique Supports n'est concernée par aucun domaine puisqu'elle se contente de marquer une méthode comme supportant l'exécution d'une transaction sans qu'il soit nécessaire de modifier le contexte transactionnel. Le domaine **Interrogation** regroupe les politiques nécessitant de consulter l'état courant de la transaction, c.-à-d. toutes les politiques autres que la politique Supports. Le domaine **Activation** regroupe les politiques nécessitant d'initier une nouvelle transaction en fonction du contexte d'exécution courant, c.-à-d. les politiques Required et Requires New. Le domaine **Interruption** regroupe les politiques nécessitant de suspendre momentanément l'exécution d'une transaction, c.-à-d. les politiques Requires New et Not Supported.

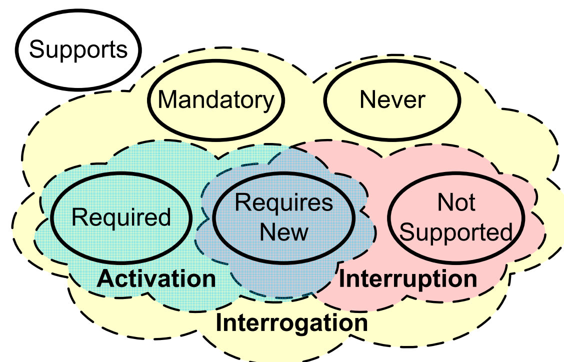


FIG. 9.1: Une organisation des politiques de démarcation transactionnelle.

La politique Supports n'appartient à aucun domaine puisqu'elle n'interagit en aucun cas avec un service de transactions. À l'opposé, la politique Requires New se trouve être à l'intersection de deux domaines puisqu'elle exécute à la fois des opérations d'interruption et d'activation de transactions. La notion de domaine constitue donc à la fois une restriction du domaine d'interaction avec le service de transactions et une simplification de son utilisation dans le cadre de la démarcation de transactions.

9.3 Défis soulevés par l'étude de la démarcation de transactions

Cette section présente les principaux défis que peut induire l'intégration de la démarcation transactionnelle dans les plates-formes orientées composants. La figure 9.2 présente les différents défis qu'il est possible d'établir lors du traitement de cette problématique.

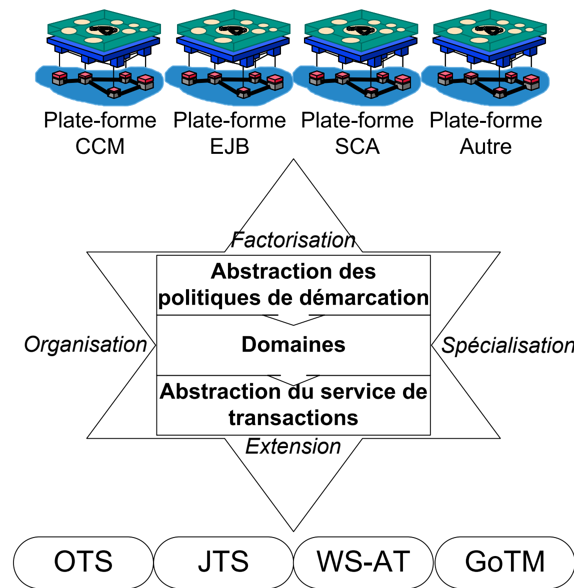


FIG. 9.2: Les défis liés à la démarcation transactionnelle.

Le premier de ces défis est la factorisation et l'abstraction des politiques de démarcation afin de pouvoir adresser un maximum de plates-formes. Le deuxième défi consiste en la personnalisation des politiques de démarcation pour un service de transactions donné. Les autres défis sont plus d'ordre architecturaux et adressent des propriétés d'organisation et d'extension des politiques de démarcation. Les sections suivantes détaillent chaque défi.

9.3.1 Abstraction de la démarcation de transactions

Beaucoup de plates-formes définissent leurs propres politiques de démarcation. Cependant, ces politiques sont mises en œuvre dans le contexte d'une plate-forme particulière. D'un point de vue transversal, ce code est donc développé de manière redondante alors qu'il devrait plutôt être factorisé. Dès lors, qui doit adresser cette factorisation ? Cela ne peut être réalisé par la plate-forme puisqu'on souhaite obtenir un aspect transverse. Cela ne peut être réalisé par le service de transactions dans la mesure où chaque politique serait définie pour un service de transactions particulier alors que les services de transactions utilisés par les plates-formes peuvent être différents.

L'abstraction des politiques de démarcation nécessite donc d'être réalisée par une tierce partie qui devra être indépendante de la plate-forme comme illustré dans la figure 9.2. Notre canevas doit fournir un ensemble de politiques répondant aux spécifications de politiques de démarcation transactionnelle définies par les plates-formes.

9.3.2 Abstraction du service de transactions

La plupart du temps, chaque plate-forme fonctionne avec un service de transactions donné et n'est pas en mesure d'intégrer un autre service de transactions de nature différente. Par exemple, les plates-formes interagissant avec des services de transactions JTS et OTS ne sont pas nombreuses. En effet, les plates-formes J2EE travaillent avec les services JTS tandis que les plates-formes CORBA interagissent plutôt avec les services de type OTS. Par exemple, la plate-forme JONAS n'est en mesure de fonctionner qu'avec le service de transactions JOTM. Aucune plate-forme n'est donc en mesure d'adresser plus d'un type de service de transactions. Ainsi, la plate-forme JONAS ne peut pas être associé à un service de transactions de type OTS.

Afin d'utiliser les politiques de transactions au dessus de différents services de transactions, une abstraction du service de transactions doit être mise en place et personnalisée dans le cadre des transactions appliquées aux composants tel que montré dans la figure 9.2.

9.3.3 Organisation des politiques de démarcation

Les politiques de démarcation peuvent être liées à un ou plusieurs domaines. Les domaines assurent la liaison entre les différents types de politiques et les services de transactions. Mais les domaines introduisent également une classification des politiques de démarcation comme cela a été mentionné dans la section 9.2.2. Cette classification se doit d'être ouverte afin que l'introduction de nouvelles politiques implique éventuellement la définition de nouveaux domaines tels que les domaines d'**association** et de **dissociation** des transactions [BP96].

Le domaine représente donc une abstraction de la démarcation comme nous avons défini une abstraction du service de transactions. Ce second niveau d'abstraction permet de personnaliser et de simplifier les fonctionnalités offertes par le service de transactions dans le contexte de la démarcation de transactions. Ainsi, les politiques deviennent plus simples à définir puisque les aspects techniques sont délégués aux domaines.

9.3.4 Intégration de nouvelles politiques de démarcation

Les politiques de démarcation introduites par les spécifications CCM et EJB ne couvrent pas l'ensemble du domaine proposé par les services de transactions. En effet, des domaines comme les transactions imbriquées ou la gestion des ressources transactionnelles ne sont pas considérées par les spécifications actuelles. Une structure ouverte doit donc être fournie afin de permettre l'intégration de nouvelles politiques comme la politique Requires New Sub définie par le canevas BOURGOGNETS [Pro02]. Cela signifie que des mécanismes pour l'ajout de politiques et d'éventuels domaines doivent être mis en place.

9.4 OTDF : le canevas de démarcation transactionnelle

En tenant compte des défis énoncés dans la section précédente, nous proposons de définir un canevas logiciel dédié aux politiques de démarcation transactionnel appelé OTDF (en anglais, *Open Transaction Demarcation Framework*). Ce canevas logiciel fournit une bibliothèque de politiques de démarcation transactionnelle. OTDF répond ainsi au problème de la démarcation transactionnelle pour de nombreux services de transactions et il est utilisable depuis n'importe quelle plate-forme applicative.

9.4.1 Présentation du canevas OTDF

Le canevas logiciel OTDF est composé de trois parties : les politiques, les domaines et l'adaptateur du service de transactions. La figure 9.3 présente une vue structurée de l'architecture d'OTDF. Le niveau des politiques considère les types de politiques de démarcation définies dans les spécifications des plates-formes (par exemple, EJB, CCM, SCA). Le niveau des adaptateurs de services de transactions regroupe les abstractions des différents services de transactions existants (par exemple, OTS, JTS, WS-AT). Le niveau des domaines définit les interactions possibles entre les politiques de démarcation et les services de transactions.

Ce canevas logiciel est suffisamment ouvert pour intégrer de nouvelles politiques de démarcation et couvrir des cas d'utilisation non supportés par les politiques de démarcation actuelles (par exemple, support des transactions étendues). En particulier, il fournit différents niveaux de mécanismes (domaines et adaptateurs) pour faciliter cette intégration. D'un point de vue technique, ce canevas logiciel fournit une représentation UML [OMG05b] qui peut être appliquée à la conception orientée objet ou composant. L'implantation composant repose sur la démarche que nous avons définie dans la partie II.

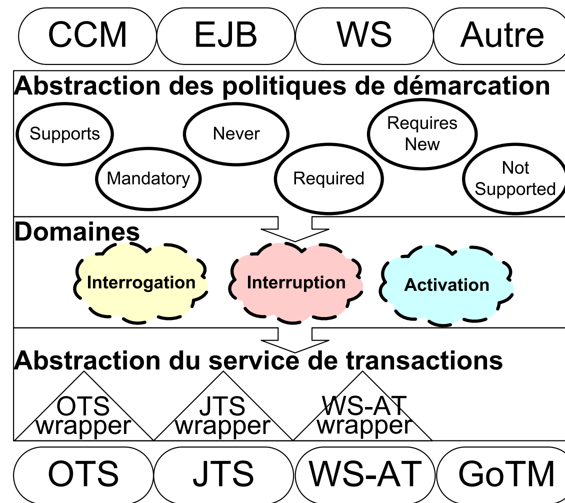


FIG. 9.3: Représentation du canevas logiciel OTDF.

9.4.2 Abstraction du service de transactions

Le premier défi technique que nous adressons dans cette section est le problème de l'abstraction du service de transactions. Généralement, le conteneur d'une plate-forme applicative est très lié à un service de transactions particulier. Parfois, plusieurs services de transactions peuvent être supportés si ceux-ci se conforment aux mêmes interfaces (par exemple, JTA, CosTransactions).

Afin de résoudre ce problème, nous appliquons le motif de conception ADAPTER [GHJV95] que nous avons déjà utilisé dans le chapitre 7 pour supporter différents standards transactionnels. Ainsi, nous pouvons convertir les interfaces de n'importe quel service de transactions vers les interfaces d'OTDF.

Le diagramme UML de classes présenté dans la figure 9.4 présente la hiérarchie d'interfaces que nous avons défini afin de fournir une abstraction modulaire des services de transactions.

Notre objectif est de définir un ensemble d'interfaces de différents niveaux de complexité pour abstraire les fonctionnalités d'un service de transactions. En effet, tous les services de transactions ne peuvent pas implanter toutes les politiques de démarcation potentielles. Par exemple, les services de transactions de type JTS seraient inadaptés aux politiques de démarcation supportant les transactions imbriquées. Il est donc nécessaire d'identifier les différences entre les services de transactions.

La complexité du service de transactions est modélisée par un ensemble d'interfaces de type `Feature`. Chaque interface `Feature` représente une fonctionnalité d'un service de transactions qui peut être indépendante des autres fonctionnalités. La figure 9.4 présente différentes fonctionnalités comme la configuration, le contrôle ou l'interrogation du service de transactions. Grâce à la granularité fine des interfaces `Feature`, l'interface du service de transactions peut être définie par composition des interfaces `Feature` associées à ses fonctionnalités.

La spécialisation des interfaces `Feature` permet également de définir les interactions possibles entre les domaines et le service de transactions avec une granularité très fine. Dans certains cas, les domaines nécessitent plusieurs fonctionnalités disponibles dans différentes interfaces `Feature`. Afin de garantir que ces interfaces sont implantées par le même service de transactions, nous définissons la notion de *scope*. Un *scope* correspond à un ensemble de fonctionnalités requises pour un domaine. Les interfaces de type `Scope` sont définies par composition des interfaces `Feature`.

Lors de l'intégration d'un nouveau modèle de transactions, seuls les éléments définissant de nouvelles fonctionnalités nécessitent d'être définis en utilisant des interfaces de type `Feature`. Si un nouveau domaine nécessite d'être défini, alors les interfaces de type `Scope` associées seront également définies. Les autres éléments sont hérités du canevas et l'abstraction du service

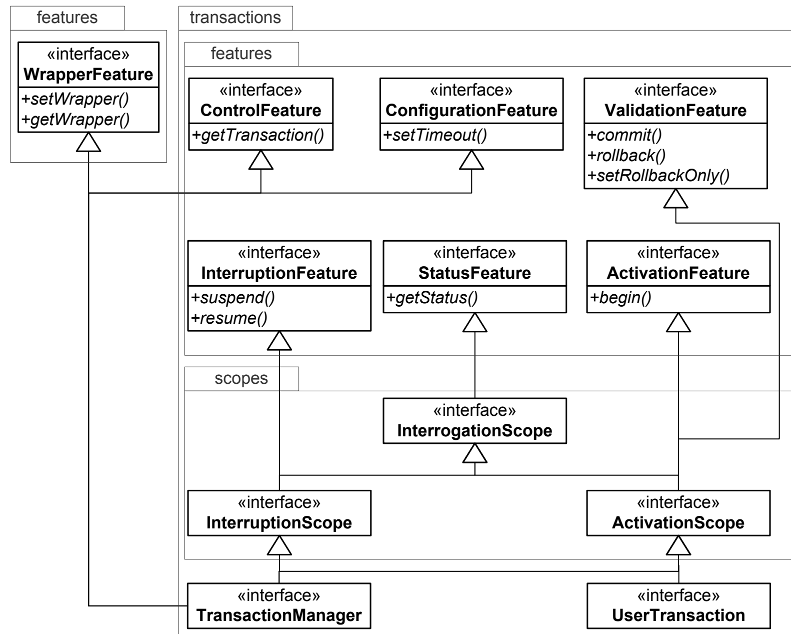


FIG. 9.4: Diagramme UML de classes de l'abstraction du service de transactions.

de transactions par composition des interfaces `Feature` requises par le nouveau modèle de transactions.

9.4.3 Abstraction de la démarcation transactionnelle

Un autre défi technique à résoudre consiste à pouvoir abstraire les politiques de démarcation. Généralement, les conteneurs des plates-formes applicatives implantent ces politiques de démarcation sans méthodologie particulière. La plupart du temps, ces implantations se contentent de vérifier la politique qu'ils doivent activer. Les traitements associés à ces politiques sont directement réalisés dans le conteneur.

Notre objectif est donc d'extraire ces politiques de démarcation du conteneur des plates-formes applicatives et ensuite de rendre ces politiques indépendantes les unes des autres. Le premier objectif peut être atteint en isolant le code de démarcation du reste du conteneur. Une solution pour atteindre le second objectif consiste à utiliser le motif de conception COMMAND [GHJV95] pour rendre ces politiques indépendantes les unes des autres.

La figure 9.5 présente l'architecture des politiques de démarcation en utilisant un diagramme UML de classes. L'interface `RequestCallController` est une interface générique de contrôle des invocations. Toute politique du canevas OTDF implante cette interface de base et ses méthodes `pre_invoke()` et `post_invoke()`.

L'interface `RequestCallContext` offre une structure de donnée générique afin de transmettre des paramètres aux politiques de démarcation. Le cycle de vie de cette structure est associé au cycle de vie de l'invocation d'une méthode sous le contrôle d'une politique de démarcation. Par conséquent, les informations concernant la méthode associée à la politique ne sont pas stockées au niveau de la politique de démarcation mais dans cette structure de données. Ainsi, les politiques sont suffisamment indépendantes pour être partagées entre plusieurs composants et conteneurs afin de réduire l'empreinte mémoire de la plate-forme.

Nous définissons différents types de politiques en liant chaque type à un domaine particulier. Ces types de politiques portent les mêmes noms que les domaines auxquels ils sont associés. Les instances de politiques héritent par conséquent des propriétés définies par le type de politique.

Grâce à cette abstraction, le canevas a les propriétés suivantes :

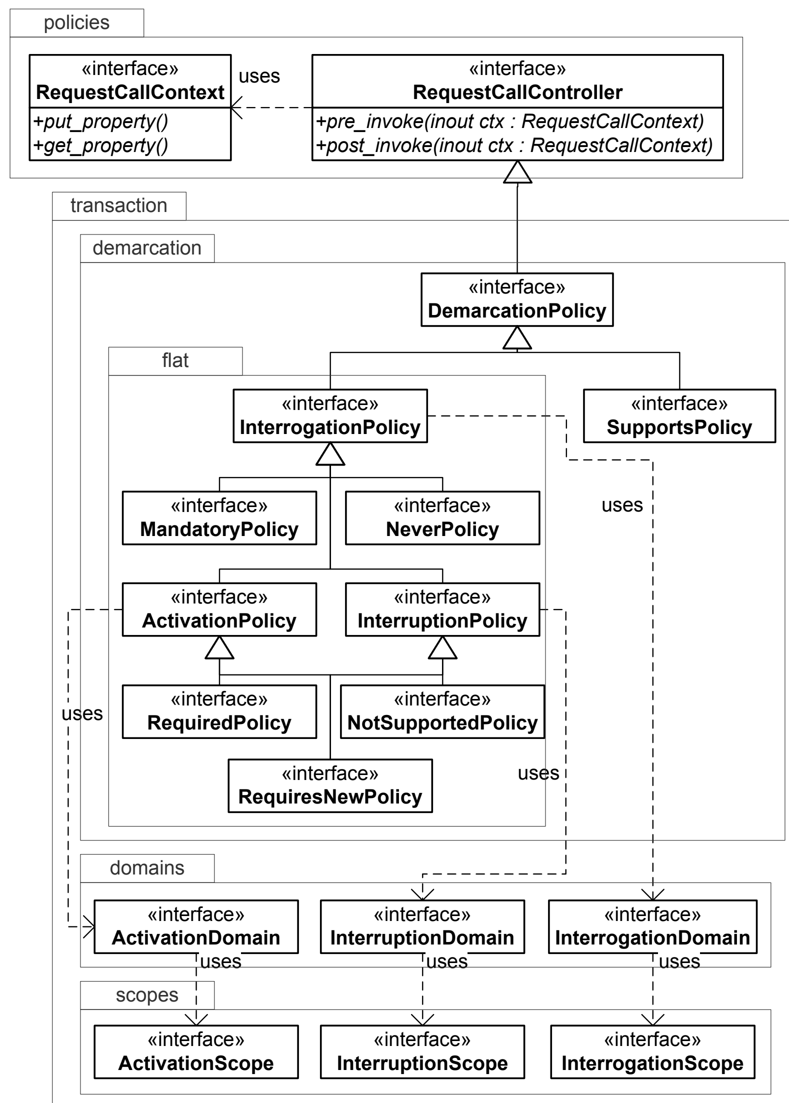


FIG. 9.5: Diagramme UML de classes de l'abstraction de la démarcation transactionnelle.

Identification : chaque traitement est associé à une seule politique.
Adaptation : les politiques de démarcation à intégrer dans la plate-forme peuvent être choisies.
Isolation : seules les politiques de démarcation requises par la plate-forme sont sélectionnées.
Factorisation : le code technique des politiques est délégué aux domaines.

9.4.4 Intégration de nouvelles politiques

La modularité du canevas OTDF offre beaucoup de flexibilité. Dès lors, il est possible d'envisager différentes intégrations telles que l'ajout d'un nouveau service de transactions ou l'ajout de nouvelles politiques de démarcation. Nous pouvons également imaginer la composition de politiques existantes afin de réaliser de nouvelles politiques.

Intégration d'un nouveau service de transactions

Une solution pour supporter de nouvelles politiques est d'intégrer un nouveau service de transactions en utilisant l'abstraction que nous avons définie. Par cette opération, les politiques disponibles dans le canevas OTDF deviennent exécutables avec le nouveau service de transactions par simple composition des politiques, des domaines et de l'abstraction du service de transactions.

Définition d'une nouvelle politique

Il est également possible de définir un nouveau type de politique de démarcation. Un exemple d'une telle politique pourrait être la politique de démarcation Requires New Sub [Pro02]. Cette politique active automatiquement une transaction imbriquée dans l'exécution d'une autre transaction. Cette politique doit ensuite être associée à un service de transactions supportant les transactions imbriquées. Le résultat est une nouvelle politique de démarcation pour tous les services de transactions supportant les transactions imbriquées.

Combinaison de politiques de démarcation existantes

Si nous observons la politique de démarcation Requires New Sub définie dans la section précédente, nous pouvons constater que sa sémantique est incomplète. En effet, son comportement n'est pas défini si la politique est activée sans contexte transactionnel. Etant donné que l'activation d'une transaction imbriquée requiert qu'au moins une transaction soit active, nous devons préciser cette sémantique. Une solution peut être d'activer une première transaction afin de pouvoir activer la transaction imbriquée. Une autre solution peut consister à forcer le client à activer une transaction avant d'invoquer une méthode métier tel que cela est défini pour la politique Mandatory.

Plutôt que d'implanter une politique de démarcation pour chaque sémantique, nous proposons d'utiliser les politiques existantes afin de raffiner la sémantique de cette politique. Ainsi, la composition des politiques Required et Requires New Sub fournit une implantation de la première sémantique. La composition des politiques Mandatory et Requires New Sub répond à la seconde sémantique. Un autre exemple est la composition des politiques Not Supported et Required afin de simuler la politique Requires New.

Ces combinaisons sont possibles avec l'utilisation d'un coordinateur qui organise l'invocation des politiques déléguées. Si ces politiques déléguées sont elles-mêmes combinées, la politique résultante est alors un arbre d'invocation de politiques élémentaires. Ce type de politique permet de préférer certaines branches d'exécution à d'autres à l'exécution en fonction du contexte d'invocation de la politique.

9.5 JOTDF : l'implantation d'OTDF en Java

Nous implantons le canevas OTDF en Java pour des raisons expérimentales. En effet, de nombreuses plates-formes applicatives (JONAS [Exe04], JBOSS [FR03], OPENEJB, EJCCM, OPENCCM [BCM04], etc.) et de nombreux services de transactions (OPENORB TS, TYREX, ARJUNATS [Lit05], JOTM [Mes03], etc.) sont disponibles en Java. Par conséquent, nous proposons de réaliser ces politiques en utilisant le langage Java. De plus, les différentes abstractions du canevas OTDF sont réalisées en utilisant notre modèle de programmation FRACLET (présenté dans le chapitre 5). L'utilisation de ce modèle de programmation offre une abstraction supplémentaire permettant de supporter différentes technologies de mise en œuvre. Ainsi, les abstractions du service de transaction, du domaine et de la politique transactionnelle peuvent supporter les technologies objet comme les modèles de composants. Cette section s'attache à présenter la réalisation de ces différentes abstractions liées à la démarcation des transactions.

9.5.1 Adaptateur de service de transactions

L'objectif de l'adaptateur est d'adapter la sémantique du service de transactions. Le listing 9.1 présente l'adaptation d'un service de transactions de type JTS pour l'intégrer au canevas OTDF.

L'interface `UserTransaction` est celle définie dans la figure 9.4 et permet d'hériter des fonctionnalités d'activation, de validation, de statut et de configuration définies par OTDF pour encapsuler un service de transactions. Les opérations de l'abstraction du service de transactions sont donc converties vers les opérations équivalentes des interfaces *Java Transaction API* (JTA). Une classe de conversion permet de convertir les statuts des transactions exprimées avec les interfaces JTA vers le formalisme défini par OTDF.

9.5.2 Domaine transactionnel

Le domaine représente le deuxième niveau d'abstraction défini par OTDF. Il représente un sous-ensemble des fonctionnalités d'un service de transactions. Le domaine simplifie et adapte la manipulation du service de transactions dans le cadre de la démarcation transactionnelle.

Le listing 9.2 présente la définition du domaine d'interruption. Les méthodes `suspend()` et `resume()` définies par l'interface `InterruptionDomain` ont accès à la structure de données `RequestCallContext` afin de pouvoir stocker différentes informations relatives au contexte d'exécution. Ici, l'information stockée est la transaction suspendue durant l'exécution de la méthode métier. Cette information sera conservée jusqu'à la fin de l'exécution de la méthode métier. L'attribut `scope` permet d'accéder au service de transactions et notamment à ses fonctionnalités d'interruption et d'interrogation.

9.5.3 Politique transactionnelle

Si les domaines et l'abstraction du service de transactions permettent d'accéder aux fonctionnalités du service de transactions sous-jacent de manière aisée, la politique capture la succession d'opérations que le conteneur doit réaliser sur le service de transactions pour réaliser la démarcation transactionnelle.

Le listing 9.3 présente l'implantation de la politique de démarcation Not Supported. Cette définition consiste à écrire le code associé aux méthodes `pre_invoke()` et `post_invoke()`. Dans le cadre de la politique Not Supported, il s'agit de suspendre l'exécution de la transaction si celle-ci est active et de restaurer l'état de la transaction après l'exécution de la méthode métier.

La définition des politiques a été simplifiée par rapport à l'implantation actuelle des politiques de démarcation dans les plates-formes applicatives. Cette définition peut être facilement réutilisée par d'autres plates-formes d'applications et avec d'autres services de transactions. De plus, la sémantique de la politique devient plus simple à comprendre de par son isolation des autres politiques et de la simplication des interfaces du service de transactions.

```
1 public class JTSUserTransactionImpl implements UserTransaction {
2   /** @requires */ protected javax.transaction.UserTransaction user-tx ;

4   public void set_transaction_timeout(int seconds) {
5     try {
6       this.user-tx.setTransactionTimeout(seconds);
7     } catch(Exception ex) { ... }
8   }
9   public Status get_status() {
10    try {
11      return JTSStatus.jts_to_status(this.user-tx.getStatus());
12    } catch (javax.transaction.SystemException ex) { ... }
13  }
14  public void begin() {
15    try {
16      this.user-tx.begin();
17    } catch (Exception ex) { ... }
18  }
19  public void commit() {
20    switch(get_status()) {
21      case Status.STATUS_ACTIVE :
22        try {
23          this.user-tx.commit();
24        } catch(Exception ex) { ... }
25        break ;
26      case Status.STATUS_MARKED_ROLLBACK :
27        rollback();
28        break ;
29  } }
30  public void rollback() {
31    try {
32      this.user-tx.rollback();
33    } catch(Exception ex) { ... }
34  }
35  public void set_rollback_only() {
36    try {
37      this.user-tx.setRollbackOnly();
38    } catch(Exception ex) { ... }
39 } }
```

LST. 9.1: Adaptateur du service de transactions JTS.

```
1 public class InterruptionDomainImpl extends InterrogationDomainImpl
2 implements InterruptionDomain {
3   /** @requires */ protected InterruptionScope scope ;

5   public void suspend(RequestCallContext ctx) {
6     try {
7       ctx.put_property("transaction_suspended", this.scope.suspend());
8     } catch(SystemException ex) { ... }
9   }
10  public void resume(RequestCallContext ctx) {
11    Transaction _tx = (Transaction) ctx.get_property("transaction_suspended") ;
12    if (_tx != null)
13      try {
14        this.scope.resume(_tx);
15      } catch(Exception ex) { ... }
16 } }
```

LST. 9.2: Illustration du domaine d'interruption.

```
1 public class NotSupportedPolicyImpl extends AbstractInterruptionPolicy
2 implements NotSupportedPolicy {
3   public void preinvoke(RequestCallContext ctx) {
4     if (this.domain.get_status(ctx) == Status.STATUS_ACTIVE)
5       this.domain.suspend(ctx) ;
6   }
7   public void postinvoke(RequestCallContext ctx) {
8     this.domain.resume(ctx) ;
9 } }
```

LST. 9.3: Illustration de la politique NotSupported.

9.5.4 Architecture d'un jeu de politiques transactionnelles

L'utilisation de notre modèle de programmation FRACLET nous offre la possibilité de décliner le canevas JOTDF sous la forme d'un assemblage de composants. Le listing 9.4 présente la description d'une architecture à base de composants réalisant l'abstraction transactionnelle. Cette architecture définit les six politiques usuellement définies par les plates-formes applicatives. Ces politiques sont définies sous la forme d'un assemblage de composants de fine granularité. Ainsi, le composant `Domains` (lignes 1–9) réifie la liste des domaines requis par les politiques de démarcation et fournis par un type de service de transactions particulier (partagé à l'aide de la définition `AutoSharing`, présentée dans le chapitre 6). Le composant `Policies` (lignes 11–19) partage automatiquement une instance de composant `Domains` entre les politiques qui nécessitent d'accéder à un service de transactions en utilisant la définition `AutoSharing`. La liste des politiques disponibles est exportée sur le composant `Policies` en utilisant la définition `AutoExport` (présentée dans le chapitre 6).

```
1 <definition name="Domains" extends="InterrogationDomain,InterruptionDomain,ActivationDomain,
2                                     AutoSharing(user-tx,JTSUserTransactionImpl)">
3   <component name="interrogation" definition="InterrogationDomainImplComposite"/>
4   <component name="interruption"  definition="InterruptionDomainImplComposite"/>
5   <component name="activation"    definition="ActivationDomainImplComposite"/>
6   <binding client="this.interrogation" server="interrogation.domain"/>
7   <binding client="this.interruption"  server="interruption.domain"/>
8   <binding client="this.activation"    server="activation.domain"/>
9 </definition>

11 <definition name="Policies" extends="AutoExport(policy,DemarcationPolicy),
12                                      AutoSharing(domain,Domains)">
13   <component name="supports"      definition="SupportPolicyImpl"/>
14   <component name="never"         definition="NeverPolicyImplComposite"/>
15   <component name="mandatory"     definition="MandatoryPolicyImplComposite"/>
16   <component name="required"      definition="RequiredPolicyImplComposite"/>
17   <component name="requires-new"  definition="RequiresNewPolicyImplComposite"/>
18   <component name="not-supported" definition="NotSupportedPolicyImplComposite"/>
19 </definition>
```

LST. 9.4: Description FRACTAL ADL des composants Domains et Policies.

La figure 9.6 fournit la représentation graphique de la description Policies. Cette description peut être intégrée dans une plate-forme applicative construite à base de composants FRACTAL.

9.5.5 Utilisation des politiques par les plates-formes applicatives

Dans cette section, nous illustrons l'utilisation de JOTDF dans une plate-forme applicative. La figure 9.7 présente un mécanisme d'interception qui peut être utilisé pour intégrer les politiques de démarcation du canevas OTDF. En effet, la plupart du temps, les plates-formes applicatives ne sont pas préparées à l'intégration d'un canevas comme OTDF et il est nécessaire d'introduire un mécanisme d'adaptation afin de pouvoir intégrer le canevas. Ici, un intercepteur `AccountInterceptor` peut être introduit en amont de la classe `AccountImpl`. Cet intercepteur appelle les méthodes `pre_invoke()` et `post_invoke()` de l'interface `RequestCallController` respectivement avant et après avoir délégué l'exécution de la méthode à l'interface `Account`.

9.6 Expérimentations

Cette section vise à démontrer que l'usage des motifs de conception COMMAND et ADAPTER n'introduit pas de surcoût à l'exécution comparé aux implantations actuelles des politiques de démarcation. Nous nous attachons également à évaluer l'empreinte mémoire du canevas OTDF.

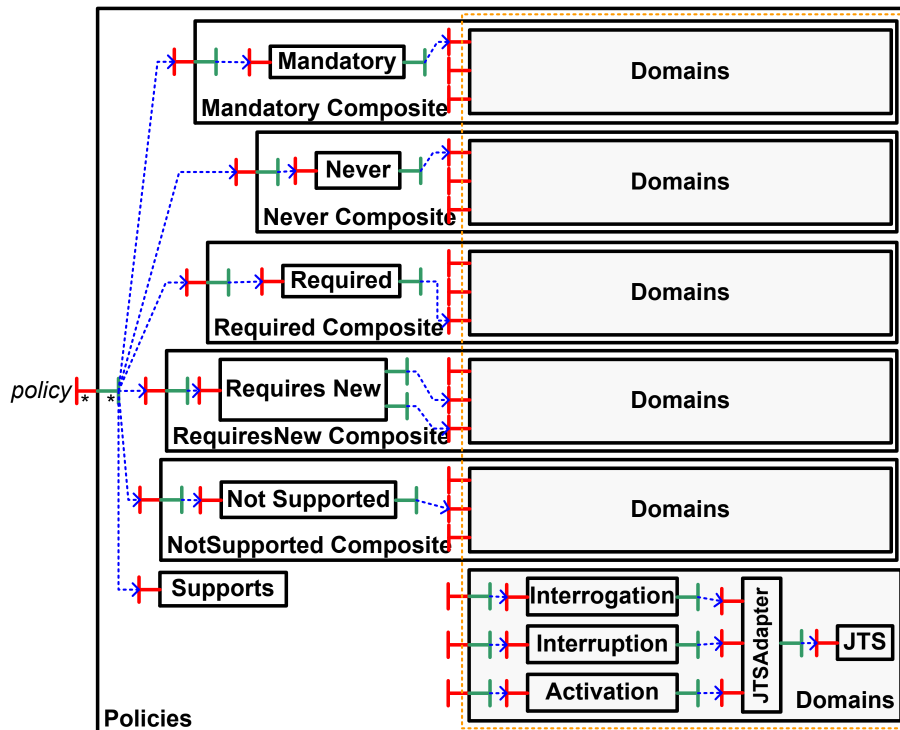


FIG. 9.6: Architecture à base de composants de la démarcation transactionnelle.

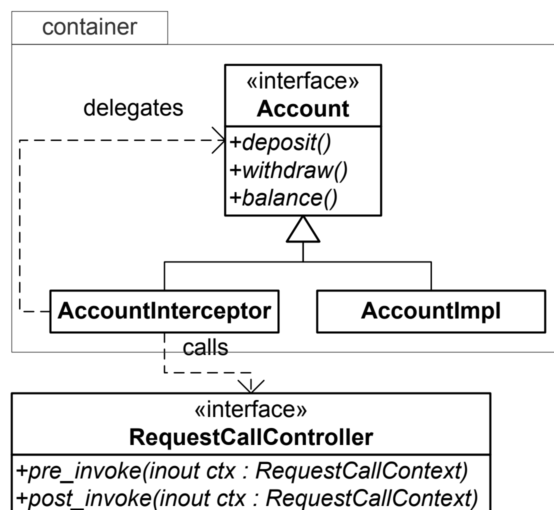


FIG. 9.7: Diagramme UML de classes de l'abstraction de la démarcation transactionnelle.

9.6.1 Contexte et scénario

Nous utilisons un Intel Pentium4 2 GHz avec 1024 Mo de RAM (DELL Optiflex GX 240) comme plate-forme d'exécution. Le système d'exploitation utilisé est un Linux Ubuntu basé sur la version $2.6.10 - 386$ du noyau et nous utilisons le JDK 1.5.0_04 comme implantation de Java. L'expérience est réalisée sur une seule machine afin d'éviter le bruit engendré par le trafic réseau. Nous comparons les canevas JOTDF au serveur d'application JOnAS version 2.5.3.

Le scénario applicatif que nous utilisons est un exemple d'application fournie avec le serveur d'application JOnAS. Il s'agit d'un simulateur de compte bancaire. Cet exemple utilise la démarcation transactionnelle du conteneur JOnAS. Nous étendons ce composant avec un composant d'évaluation de performances. Ce composant utilise le composant banque pour réaliser 10 000 invocations dans un contexte transactionnel et 10 000 invocations sans contexte transactionnel.

Nous évaluons dans un premier temps la version traditionnelle de JOnAS puis une version de JOnAS intégrant le canevas JOTDF. Nous souhaitons évaluer le coût d'invocation des méthodes métiers du composant banque dans six situations différentes. Chaque situation correspond à l'utilisation de l'une des six politiques disponibles dans JOTDF.

9.6.2 Évaluation mémoire

Une évaluation statique de l'empreinte mémoire de JOTDF peut être facilement réalisée. Cette évaluation, présentée dans le tableau 9.2, consiste à dénombrer le nombre de classes nécessaires à implanter avec le canevas OTDF en comparant cette implantation à une implantation naïve d'un tel canevas.

La variable X représente le nombre de classes requises pour implanter les politiques de démarcation tandis que la variable Y représente le nombre de classes requises pour intégrer les services de transactions. La variable Z correspond au nombre de domaines requis par les différentes politiques d'OTDF.

	Démarcation transactionnelle	
	sans JOTDF	**avec JOTDF**
Classes	X TD $\times$ Y TM	X TD + Z D + Y TM
Exemple (6 politiques, 3 TM)	6 TD $\times$ 3 TM = 18 classes	6 TD + 3 D + 3 TM = 12 classes
Nouvelle politique (TD)	Y classes	1 classe
Nouveau service (TM)	X classes	1 classe

TAB. 9.2: Évaluation de la taille de JOTDF.

Une implantation classique d'un tel canevas donne une empreinte mémoire beaucoup plus importante que celle obtenue avec JOTDF. En ce qui concerne les extensions des politiques de démarcation, l'intégration d'une nouvelle politique telle que la politique Requires New Sub dépend du nombre de services de transactions supportés dans une implantation naïve. Dans JOTDF, cette intégration correspond au développement d'une seule classe. De la même façon, l'extension du canevas pour la prise en compte d'un nouveau service de transactions dépend du nombre de politiques fournies par le canevas quand il ne représente qu'une classe dans le canevas JOTDF. JOTDF propose une évolution linéaire face aux extensions du canevas alors que l'évolution de l'implantation naïve de ces mêmes extensions serait exponentielle.

De plus, les politiques de démarcation de JOTDF peuvent facilement être partagées entre les composants et les conteneurs de la plate-forme applicative. Par conséquent, indépendamment du nombre de composants hébergés par la plate-forme d'application, JOTDF n'utilise qu'une seule instance des politiques du canevas. À l'heure actuelle, la taille de la démarcation transactionnelle est liée au nombre de conteneurs déployés sur la plate-forme applicative.

9.6.3 Évaluation de performances

Dans cette section, nous nous attachons à évaluer le coût d'exécution du canevas JOTDF. Cette évaluation et sa comparaison à une implantation traditionnelle de la démarcation transactionnelle est présentée dans le tableau 9.3.

Politiques	JOnAS	JOnAS & JOTDF	Évolution
Supports	17,697 sec	17,569 sec	-0,72 %
Not Supported	18,324 sec	18,302 sec	-0,12 %
Required	31,013 sec	30,963 sec	-0.16 %
Requires New	42,869 sec	42,832 sec	-0,09 %
Never	91,869 sec	90,940 sec	-1,01 %
Mandatory	97,419 sec	97,027 sec	-0,40 %
		Moyenne	-0,42 %

TAB. 9.3: Évaluation du coût d'exécution des politiques de JOTDF.

Les résultats présentés dans le tableau 9.3 prouvent que la structuration et la modularité du canevas OTDF n'introduisent pas de surcoût à l'exécution comparé à l'implantation de la démarcation transactionnelle telle qu'elle est réalisée par les plates-formes applicatives actuelles.

L'utilisation du motif de conception COMMAND offre un coût d'exécution uniforme. En effet, contrairement aux pratiques actuelles, le coût d'exécution d'une politique de notre implantation est totalement indépendante des autres politiques.

9.7 Conclusion

Ce chapitre a présenté un canevas logiciel pour adresser le problème de l'intégration de la démarcation transactionnelle dans les plates-formes applicatives. Pour répondre au problème de l'intégration de la démarcation transactionnelle face à l'accroissement du nombre de plates-formes applicatives et de services de transactions, nous avons proposé le canevas OTDF (en anglais, *Open Transaction Demarcation Framework*). Ce canevas offre un moyen de capitaliser la fonction de démarcation transactionnelle indépendamment des plates-formes applicatives et des services de transactions. La structuration du canevas en trois couches (politique/domaine/abstraction du service) facilite la définition et l'extension des politiques de démarcation.

L'implantation en Java du canevas OTDF prouve que l'ingénierie de cette fonctionnalité n'introduit pas de surcoût à l'exécution tout en offrant des propriétés de modularité, d'indépendance et d'extensibilité que les approches actuelles ne proposent pas.

Les travaux présentés dans ce chapitre donné lieu à deux publications internationales [RM03, PRC03].

90% of a programmer errors come from data from other programmers.

Murphy's Law

Chapitre 10

ATS : l'adaptation des standards transactionnels lors de la conception

Sommaire

10.1 Introduction **180**
10.2 Problématique de la composition **180**
10.3 Conception du service de transactions adapté **181**
10.3.1 Présentation du service de transactions adapté 181
10.3.2 Analyse des fonctions 181
10.3.3 Définition des stratégies 185
10.3.4 Composition du service de transactions adapté 187
10.3.5 Scénario d'utilisation du service de transactions 188
10.4 Mise en œuvre et évaluation du service transactions adapté **189**
10.4.1 Éléments d'implantation avec GoTM 189
10.4.2 Évaluation des performances 190
10.5 Travaux connexes **192**
10.5.1 Interopérabilité des standards transactionnels 192
10.5.2 Adaptation des services de transactions 192
10.6 Conclusion **193**

Ce chapitre s'intéresse au problème de la composition des services *de transactions qui sont soumis à différents standards. Parmi ces standards, il est possible de citer les spécifications* Object Transaction Service, Java Transaction Service, *ou* Web Services Atomic Transaction. *Cependant, des spécifications comme* Web Services Atomic Transaction *ont pour objectif d'adapter les standards actuels à la plate-forme* Web Services. *Ce mécanisme d'encapsulation introduit une complexité supplémentaire dans le système tout en masquant les spécificités des standards transactionnels existants. Lors de la composition d'applications hétérogènes, il s'avère que les services de transactions sous-jacents ne peuvent pas être composés de manière transparente. Dans ce chapitre, nous présentons une approche pour construire un service de transactions adapté, appelé ATS (en anglais,* Adapted Transaction Service*), capable de supporter plusieurs standards transactionnels simultanément.*
L'objectif d'ATS est de faciliter la composition des standards transactionnels. Après avoir survolé les standards transactionnels Object Transaction Service, Java Transaction Service, *et* Web Services Atomic Transaction, *nous présentons l'approche que nous appliquons pour construire une implémentation de ATS en utilisant le canevas logiciel* GoTM.

10.1 Introduction

Ces dernières années, le nombre de standards transactionnels a constamment augmenté. Parmi ces standards, les spécifications *Object Transaction Service* (OTS) [OMG03], *Java Transaction Service* (JTS) [Che99], ou *Web Services Atomic Transaction* (WS-AT) [CCF+05a] consituent les références des standards transactionnels. La tendance actuelle en matière de définition de standard consiste à étendre les standards existants en les encapsulant avec de nouvelles interfaces. Par exemple, la spécification WS-AT étend JTS, elle-même étendant le standard OTS. Cependant, cette approche a tendance à introduire une complexité supplémentaire dans chaque couche d'encapsulation et elle perd les spécificités offertes par les standards existants. Lorsque des applications hétérogènes sont composées, leurs services de transactions respectifs ne peuvent pas être composés de manière transparente. Cet inconvénient nous encourage à proposer une approche pour faciliter la composition des standards transactionnels.

Ce chapitre présente notre approche pour construire un service de transactions adapté (*Adapted Transaction Service* [ATS]) et son implémentation à base de composants de fine granularité. L'ATS compose plusieurs standards transactionnels simultanément et assure la compatibilité de leurs différentes fonctions. Pour concevoir un ATS, nous analysons les interfaces des standards transactionnels considérés et nous identifions les fonctions fournies par ces standards. Chaque fonction est ensuite dérivée en une ou plusieurs stratégies selon les différentes sémantiques requises par les standards. L'ATS est obtenu par composition des stratégies avec les adaptateurs. Ces adaptateurs assurent la compatibilité entre les fonctions et les interfaces imposées par les standards transactionnels. Nous utilisons le canevas logiciel GoTM pour implémenter un ATS composant les standards transactionnels CORBA, Web Services et Java.

La suite du chapitre est organisée de la manière suivante. La section 10.2 présente la problématique de la composition des standards transactionnels. La section 10.3 introduit la conception d'un service de transactions adapté à plusieurs standards transactionnels. La section 10.4 détaille la réalisation du service de transactions en utilisant le canevas GoTM (présenté dans le chapitre 7). La section 10.5 compare notre proposition avec les travaux connexes à cette problématique. Enfin, la section 10.6 conclut.

10.2 Problématique de la composition

Nous illustrons le problème relatif à la composition des standards transactionnels par un exemple mettant en jeu une application de réservation de vols et une autre application de réservation d'hôtels comme illustré dans la figure 10.1.

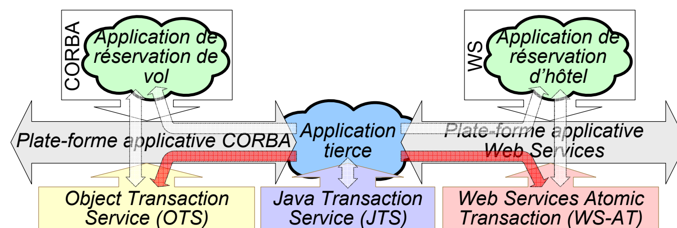


FIG. 10.1: Problème de la composition des standards transactionnels.

Chacune de ces applications est hébergée sur différentes plates-formes distribuées (CORBA et Web Services). Ces plates-formes supportent leur propre standard transactionnel. La plate-forme CORBA supporte OTS (*Object Transaction Service*) et la plate-forme Web Services supporte WS-AT (*Web Services Atomic Transaction*). L'application tierce composant les deux applications de réservation utilise JTS (*Java Transaction Service*). Cette application tierce peut interagir avec les deux autres applications en utilisant les fonctionnalités d'accès distant disponibles dans chacune des plates-formes associées.

Si cette architecture supporte l'interaction d'applications hétérogènes, le contexte transactionnel[1] n'est pas propagé implicitement d'une application à une autre. Ceci signifie que les transactions prises en charge par le service WS-AT ne peuvent pas coopérer avec les transactions prises en charge par le service OTS. De plus, l'application tierce n'est pas en mesure de synchroniser l'exécution des transactions en cours d'exécution dans les deux autres services de transactions.

Une telle synchronisation requiert que l'application tierce contrôle tous les services de transactions en utilisant trois types d'interfaces différentes. Ainsi, l'application tierce doit démarrer explicitement une nouvelle transaction avec chaque service de transactions. Cependant, lorsque les transactions doivent être validées, l'application tierce doit trouver une solution pour coordonner les protocoles de validation des trois services de transactions. Les solutions existantes emploient les mécanismes de compensation pour supporter la coordination de plusieurs transactions [CCF+05c]. Cependant, des actions de compensation ne peuvent pas toujours être définies (par exemple, compenser l'envoi d'un *email*). Par conséquent, les mécanismes de compensation présentent des limites dans certaines situations. De plus, la définition d'un mécanisme de coordination au niveau de l'application tierce tend à mélanger le code relatif au métier de l'application avec celui relatif à la coordination des transactions.

Dans ce chapitre, nous proposons une approche pour prendre en compte ce problème d'hétérogénéité des standards transactionnels existants. Plutôt que de proposer l'utilisation d'un langage unifié ou un nouveau standard transactionnel, nous souhaitons construire un service de transactions adapté capable de supporter plusieurs standards transactionnels simultanément. Par conséquent, l'application tierce utilise uniquement les interfaces de son service de transactions (en l'occurence JTS) et les transactions sont automatiquement coordonnées avec les autres standards transactionnels. Grâce à cette approche, des applications patrimoniales peuvent être composées de façon transparente d'un point de vue transactionnel.

10.3 Conception du service de transactions adapté

Cette section présente l'approche que nous avons définie pour construire un service de transactions adapté. Après avoir présenté l'architecture du service de transactions adapté, nous détaillons les étapes d'analyse des fonctions, de définitions des stratégies et de composition de ces dernières nécessaires à la construction du service de transactions adapté.

10.3.1 Présentation du service de transactions adapté

La figure 10.2 présente un ATS supportant les standards transactionnels OTS, JTS et WS-AT. Un ATS est généralement composé d'un ensemble d'adaptateurs et d'un service de transactions commun (*Common Transaction Service* [CTS]). Le CTS est une implémentation d'un service de transactions générique. Celui-ci regroupe toutes les fonctions supportées par l'ATS et requises par les trois standards transactionnels. Les adaptateurs assurent la compatibilité avec les différents standards supportés. Ceux-ci sont responsables de la conversion des opérations réalisées dans un standard transactionnel particulier vers les fonctions fournies par le CTS.

Cette architecture modulaire peut être facilement étendue afin de supporter de nouveaux standards transactionnels. Dans ce cas, le contenu du CTS peut être adapté pour prendre en compte de nouvelles fonctionnalités requises par un standard.

10.3.2 Analyse des fonctions

Une fonction se compose d'un ensemble d'opérations liées par leur sémantique. Ainsi, nous faisons l'hypothèse qu'un service de transactions repose sur trois fonctions essentielles : *Status*, *Coordination* et *Participants*. Ensuite, nous associons ces trois fonctions aux interfaces définies par

[1] contexte transactionnel : information relative à l'exécution d'une transaction.

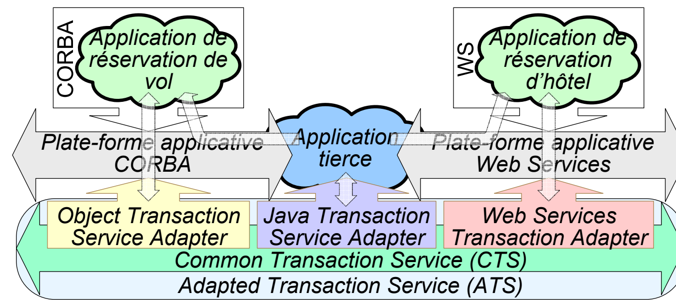


FIG. 10.2: Illustration du service de transactions adapté.

chaque standard transactionnel. En particulier, nous nous intéressons aux interfaces *CosTransactions API*, *Java Transaction API* et *Atomic Transaction Services* définies respectivement par les standards transactionnels OTS, JTS et WS-AT. Cette analyse nous permet également de confirmer que trois fonctions sont suffisantes pour caractériser un service de transactions.

Analyse de JTA

JTA définit un ensemble d'interfaces Java afin d'offrir un support transactionnel aux applications Java. La figure 10.3 liste ces différentes interfaces et les associe à l'une des trois fonctions que nous avons identifié.

- L'interface Status définit les constantes représentant les différents états disponibles (`NO_TRANSACTION`, `STATUS_ACTIVE`, `STATUS_COMMITED`, `STATUS_ROLLEDBACK`, etc.).
- L'interface Transaction réifie une transaction à l'exécution et fournit une méthode `getStatus()` permettant de consulter le statut courant d'une transaction en retournant l'une des constantes définies dans l'interface Status. Les méthodes `setRollbackOnly()`, `rollback()` et `commit()` permettent de contrôler la terminaison d'une transaction et l'exécution du protocole de validation en deux phases. Les méthodes `enlistResource()`, `delistResource()` et `registerSynchronization()` permettent d'associer et de dissocier les participants d'une transaction en cours d'exécution.
- Les interfaces UserTransaction et TransactionManager correspondent à deux façades définies pour manipuler l'interface Transaction en fonction du rôle des entités appelantes. Ainsi, l'interface UserTransaction est destinée à être utilisée par l'application tandis que l'interface TransactionManager est plutôt manipulée par le serveur d'applications.
- Les interfaces XAResource et Synchronization définissent deux types de participants pour une transaction. Les participants de type XAResource sont notifiés de l'évolution du cycle de vie de la transaction et ils sont consultés lors de l'exécution du protocole de validation en deux phases. Les participants de type Synchronization sont simplement notifiés du début et de l'issue du protocole de validation en deux phases.

À partir de notre analyse, nous établissons les dépendances entre les interfaces. Ces dépendances sont marquées par les relations utilisant le stéréotype UML *uses* [OMG05b]. Par exemple, l'interface Transaction dépend des interfaces Status, Synchronization et XAResource. Cette dépendance est dûe au fait que certaines opérations requièrent les interfaces Synchronization et XAResource ou fournissent l'interface Status.

Ensuite, nous classons les interfaces JTA selon la sémantique indiquée par leurs opérations. Les fonctions identifiées sont repérées par le stéréotype UML *function* dans la figure 10.3. Par exemple, la sémantique des interfaces UserTransaction, Transaction, et TransactionManager font référence au protocole de validation en deux phases [SARM05] et sont par conséquent associées à la fonction *Coordination*. En suivant la même approche pour les autres interfaces de JTA, la fonction *Status* est associée à l'interface Status et la fonction *Participants* est associée aux interfaces Synchronization et XAResource.

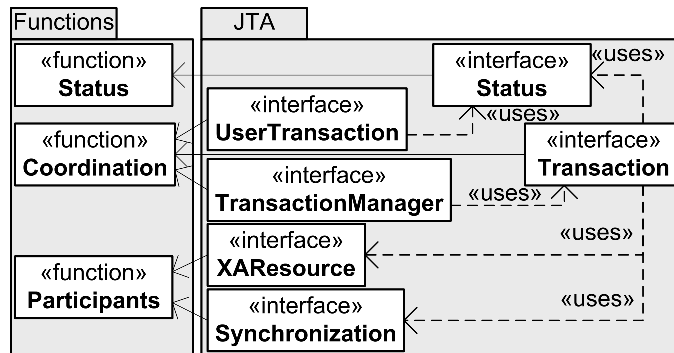


FIG. 10.3: Analyse des interfaces Java JTA.

Analyse de CosTransactions

Les interfaces *CosTransactions* sont définies en utilisant le langage de description d'interfaces de l'OMG (en anglais, *IDL pour Interface Description Language*) [OMG04]. Ce langage répond en partie à l'interopérabilité d'applications réparties écrites dans des langages de programmation différents. La figure 10.4 nous permet d'identifier les fonctions mises en jeu dans le service de transactions OTS.

- L'interface Status définit les constantes représentant les différents états disponibles (`NoTransaction`, `StatusActive`, `StatusCommitted`, `StatusRolledback`, etc.).
- Les interfaces Control, Coordinator et Terminator réifient une transaction à l'exécution. L'interface Control permet d'accéder aux interfaces Coordinator et Terminator. L'interface Coordinator définit les méthodes `enlist_resource()`, `delist_resource()` et `register_synchronization()` pour permettre d'associer et de dissocier les participants, et une méthode `get_status()` permettant de consulter le statut courant d'une transaction en retournant l'une des constantes définies dans l'interface Status. L'interface Terminator fournit les méthodes `set_rollback_only()`, `rollback()` et `commit()` qui permettent de contrôler la terminaison d'une transaction et l'exécution du protocole de validation en deux phases.
- Les interfaces TransactionFactory et Current correspondent respectivement aux modes d'utilisation direct et indirect du service de transactions. Ainsi, l'interface Current permet de manipuler de façon transparente une transaction. L'interface TransactionFactory permet de créer explicitement de nouvelles instances de transactions représentée par l'interface Control.
- Les interfaces Resource, Synchronization et SubtransactionAwareResource définissent trois types de participants pour une transaction. Les participants de type Resource sont consultés lors de l'exécution du protocole de validation en deux phases. Les participants de type Synchronization sont simplement notifiés du début et de l'issue du protocole de validation en deux phases. Les participants de type SubtransactionAwareResource supportent l'exécution de transactions imbriquées.
- L'interface RecoveryCoordinator est utilisée en cas de reprise sur faute pour synchroniser le service de transactions avec les différents gestionnaires de ressources.

Après avoir établi les dépendances entre les interfaces CosTransactions, nous identifions les fonctions requises. Il s'agit des fonctions de *Coordination* pour l'interface Terminator car celle-ci contrôle directement l'exécution du protocole de validation en deux phases. La fonction *Status* est associée à l'interface Status tandis que la fonction *Participants* représente les interfaces Resource, SubTransactionAwareResource et Synchronization correspondant aux trois types de participants qui peuvent être associés à une transaction OTS.

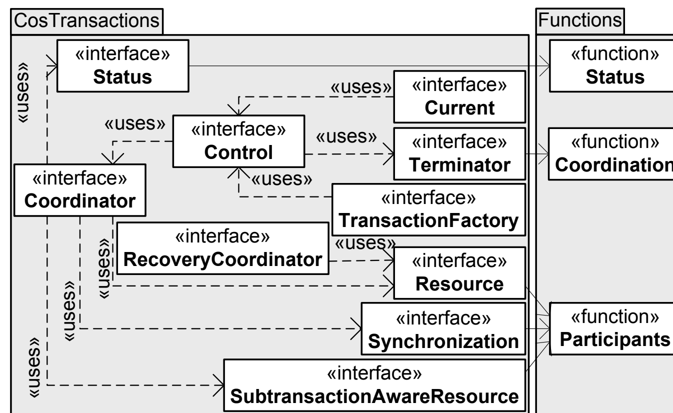


FIG. 10.4: Analyse des interfaces IDL CosTransactions.

Analyse de WS Atomic Transaction

Les interfaces du standard *Web Services Atomic Transaction* (WS-AT) sont définies en utilisant le langage de description des *Web Services* (*Web Services Description Language* [WSDL]) [CMRW06]. Ce langage s'apparente aux langages de description d'interfaces et il fournit une syntaxe XML pour décrire les interactions possibles avec un *Web Service*. La figure 10.5 adopte une syntaxe graphique pour représenter les grandes fonctionnalités des spécifications *Web Services Atomic Transaction* [CCF+05a] et *Web Services Coordination* [CCF+05a].

- Le *Web Service* Activation permet de créer un nouveau contexte de coordination. La sémantique de ce contexte n'est pas fixée et peut servir à d'autres services que le service de transactions.
- Le *Web Service* Registration permet d'associer des participants au contexte de coordination créé par le *Web Service* Activation.
- Le *Web Service* 2PC permet d'initier le processus de validation en deux phases.
- Le *Web Service* PhaseZero permet de notifier les participants de l'initiation du protocole de validation.
- Le *Web Service* Completion permet d'inclure un participant dans le protocole de validation de la transaction.
- Le *Web Service* CompletionWithAck permet également d'inclure un participant dans le protocole de validation de la transaction mais il force le coordinateur à mémoriser l'issue tant qu'il n'a pas reçu l'accusé de réception du participant.
- Le *Web Service* OutcomeNotification permet simplement de notifier les participants de l'issue de la transaction.

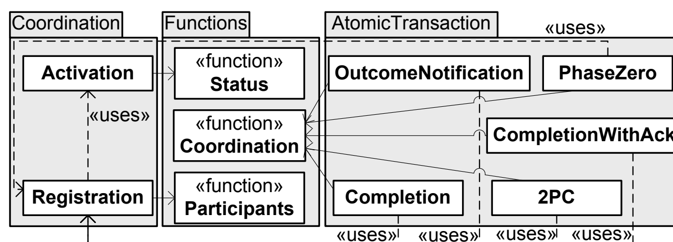


FIG. 10.5: Analyse des interfaces WSDL Atomic Transaction.

Le standard *Web Services Atomic Transaction* a la particularité de reposer sur un standard générique de coordination de *Web Services* dénommé *Web Services Coordination* [Arj03b]. Ce standard définit les fonctionnalités nécessaires pour gérer un ensemble de participants et les notifier

de différents types d'événements [Arj03a]. Le standard *Web Services Atomic Transaction* s'appuie sur ces fonctionnalités pour fournir la propriété d'atomicité transactionnelle à un ensemble de *Web Services*. La résolution des dépendances et l'identification des fonctions pour ces deux standards associe la fonction de *Status* au *Web Service* Activation, la fonction de *Participants* au *Web Service* Registration et la fonction *Coordination* aux *Web Services* OutcomeNotification, PhaseZero, CompletionWithAck, 2PC et Completion.

10.3.3 Définition des stratégies

Dans cette section, l'objectif est de définir exhaustivement la sémantique qui doit être attachée aux fonctions que nous avons identifié. Cette sémantique (appelée stratégie) est imposée par les différents standards transactionnels que nous considérons. La figure 10.6 présente les différentes stratégies associées aux fonctions *Status*, *Coordination* et *Participants*.

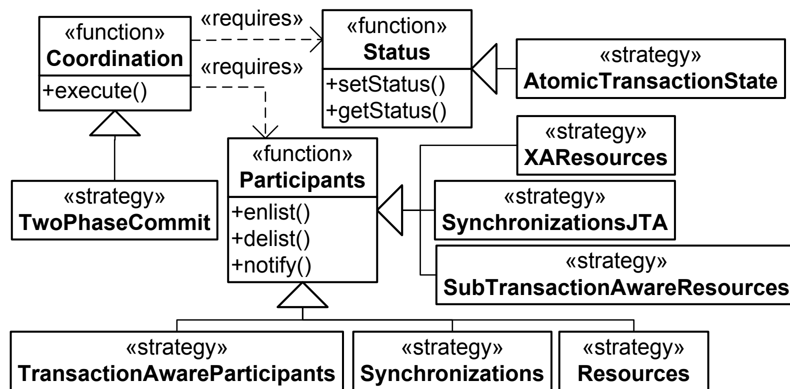


FIG. 10.6: Présentation des stratégies disponibles.

La fonction *Status* est associée à la stratégie `AtomicTransactionState`. Celle-ci décrit le comportement d'une transaction atomique. La fonction *Coordination* est associée à la stratégie `TwoPhaseCommit`. Cette stratégie décrit le comportement d'un protocole de validation à deux phases. La fonction *Participants* est associée à six stratégies. Les stratégies `XAResources` et `SynchronizationsJTA` sont dédiées au standard JTA. Les stratégies `SubTransactionAwareResources`, `Resources` et `Synchronizations` sont définies pour répondre au standard OTS. Finalement, la stratégie `TransactionAwareParticipants` gère les participants associés au standard WS-AT. Chacune de ces stratégies est décrite à l'aide de plusieurs modèles dédiés de haut niveau (présentés dans le chapitre 8).

Stratégie *Two-Phase Commit*

La stratégie *Two-Phase Commit* représente une implémentation du protocole de validation en deux phases. Ce protocole définit en particulier les échanges de messages réalisés entre la transaction et ses participants pour aboutir à une décision de validation ou d'abandon commune. La figure 10.7 utilise le diagramme de séquences UML pour décrire le protocole de validation en deux phases.

Le protocole de validation en deux phases consiste à envoyer un premier message `prepare` à tous les participants de la transaction. Chaque participant répond à ce message avec un message de vote précisant l'issue choisie par le participant (`vote-commit` ou `vote-abort`). Si tous les votes collectés par la stratégie sont du type `vote-commit`, celle-ci émet dans ce cas un message `commit` aux participants. Dans le cas contraire, c'est un message `abort` qui est envoyé aux participants. Après avoir reçu l'un de ces messages et avoir effectué les traitements associés, chaque participant valide le message par un accusé de réception de type `ack`. La figure 10.7 présente la description de la version traditionnelle du protocole de validation en deux phases [Gra78].

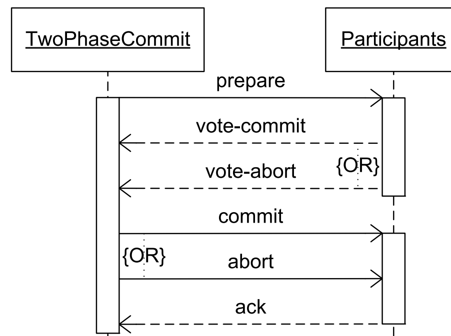


FIG. 10.7: Description de la stratégie *2PC*.

Néanmoins, l'utilisation du diagramme de séquences UML pour décrire les protocoles de coordination nous permet de considérer des protocoles tels que les protocoles *Two-Phase Commit Presumed Abort* (2PC-PA) ou *Two-Phase Commit Presumed Commit* (2PC-PC) [MLO86]. Le chapitre 11 présente une expérience d'adaptation dynamique de ces protocoles de validation en deux phases.

Stratégie *Synchronizations JTA*

La fonction *Participants* peut être dérivée en six stratégies différentes. Pour chacune de ces stratégies, nous utilisons le formalisme Evénément/Condition/Action (ECA) [CC95] pour exprimer le comportement des différents types de participants face aux événements émis par le service de transactions. Les détails relatifs à l'utilisation de ce modèle ont été présentés dans le chapitre 8. Dans cette section, nous détaillons la description du comportement des participants de type *Synchronization* définis par le standard JTA. Le listing 10.1 présente les règles ECA définies pour ce type de participants.

```
1 @Rules(name="SynchronizationsJTA")
2 public class Synchronizations {
3   @Global Status status;

5   @On(event="prepare")
6   void prepare(Synchronization synchronization) {
7     synchronization.beforeCompletion();
8   }
9   @On(event="acknowledge")
10  void ack(Synchronization synchronization) {
11    synchronization.afterCompletion(status.getStatus());
12 } }
```

LST. 10.1: Description des participants de type *Synchronization*.

Pour chaque message supporté par le type de participant, des actions sont réalisées en réponse à l'événement correspondant à l'émission de l'un des messages. Quant à la partie action de la règle, celle-ci consiste en une séquence d'instructions à réaliser sur les participants. Ainsi lorqu'on considère les participants de type *Synchronization* tels qu'ils sont définis dans le standard JTA, l'émission du message `prepare` implique un appel de la méthode `beforeCompletion()` sur tous les participants de type *Synchronization* associés à la transaction. L'émission des messages `commit` (respectivement `abort`) se traduit par l'appel de la méthode `afterCompletion()` sur ces mêmes participants en utilisant la valeur `Status.STATUS_COMMITTED` (respectivement `Status.STATUS_ROLLEDBACK`) obtenus à partir de l'appel de la méthode `getStatus()`.

Stratégie *Atomic Transaction State*

La fonction *Status* est dérivée avec la stratégie *Atomic Transaction State*. Cette stratégie identifie les états caractéristiques d'une transaction utilisant un modèle de transactions atomiques. Ce modèle est caractérisé par les deux issues possibles pour toute transaction : le succès (en anglais, *commit*) ou l'abandon (en anglais, *abort ou rollback*). Nous utilisons le diagramme UML d'états, présenté dans la section 8.4, pour décrire le comportement de l'automate représentant les états et les transitions possibles d'une transaction utilisant le modèle atomique.

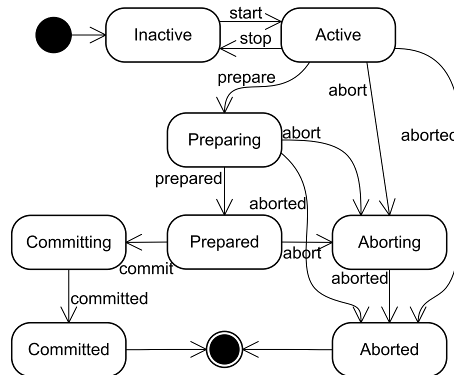


FIG. 10.8: Description de la stratégie *Atomic Transaction State*.

Comme illustré dans la figure 10.8, une transaction atomique débute dans l'état Inactive. Dans cet état, la transaction a été créée par le service de transactions mais elle n'est pas encore en cours d'exécution. Lors du déclenchement de la transition start, la transaction passe dans l'état Active. Le passage dans cet état marque le début de la transaction. Toute transaction peut alors être suspendue et reprise en utilisant les transitions stop et start. Lors de l'activation de la transition prepare, la transition passe dans l'état Preparing. Cet état correspond à la première phase du protocole de validation en deux phases. La transaction peut également être abandonnée (état Aborting) ou oubliée (état Aborted) en utilisant les transitions abort ou aborted depuis les états Active ou Preparing. Cette décision est alors irrévocable et empêche définitivement la validation de la transaction. L'activation de la transition prepared indique que tous les participants ont répondu à la première phase du protocole et que la décision quant à l'issue de la transaction peut être prise. Dès lors les transitions commit ou abort peuvent être activées pour que l'automate migre respectivement vers les états Committing ou Aborting et que les messages `commit` ou `abort` puissent être envoyés aux participants. Le passage dans l'état Committed (respectivement Aborted) est activé par la transition committed (respectivement aborted) correspondant à la réception de tous les accusés de réception des participants à la transaction. Chacun de ces états est considéré comme un état final et stable de la transaction. Ce diagramme UML d'état peut également être étendu pour intégrer de nouveaux états comme un état Compensating correspondant à la compensation d'une transaction validée [OMG05a].

10.3.4 Composition du service de transactions adapté

L'objectif de cette section est d'assembler le service de transactions commun (*Common Transaction Service* [CTS]). En suivant les dépendances identifiées lors de l'analyse des standards transactionnels, il est possible de construire le CTS en sélectionnant les stratégies à utiliser. Le CTS correspond à un service de transactions générique capable de fournir les fonctionnalités requises par les trois standards transactionnels que nous considérons. Afin de faciliter la composition des stratégies utilisées pour construire le CTS, nous utilisons le paradigme de bus logiciel [Esk99]. Le bus logiciel propose les fonctionnalités requises pour intégrer une stratégie dans le CTS et connecter celle-ci avec les autres stratégies dont elle dépend (conformément aux dépendances

liant leurs fonctions respectives). En particulier, il facilite la propagation des messages commit ou abort entre la stratégie *TwoPhaseCommit* et la stratégie *SynchronizationsJTA*.

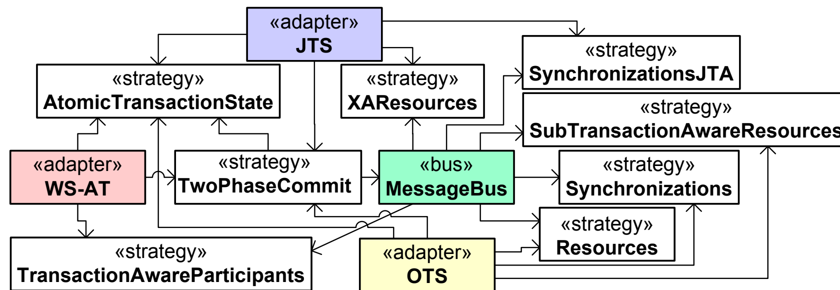


FIG. 10.9: Description de la composition du service de transactions adapté.

La figure 10.9 illustre cet assemblage de stratégies pour la construction du CTS conforme aux standards JTS, OTS et WS-AT. L'élément MessageBus représente le bus logiciel utilisé pour assembler le CTS et propager les messages entre les différentes stratégies. Les associations lient une stratégie au bus logiciel ou à une autre stratégie afin de résoudre les dépendances de fonctions requises/fournies.

La compatibilité avec les interfaces requises par les différents standards transactionnels est réalisée via la définition d'adaptateurs d'interfaces. Ces adaptateurs sont ensuite liés aux stratégies requises par leur standard. L'union des adaptateurs, des stratégies et du bus logiciel forme le service de transactions adapté (ATS). La figure 10.9 présente par conséquent l'ATS supportant simultanément les standards transactionnels JTS, OTS et WS-AT.

10.3.5 Scénario d'utilisation du service de transactions

L'objectif de cette section est d'illustrer sur l'exemple présenté en début de chapitre (voir la figure 10.2) le fonctionnement de l'ATS.

1. L'application tierce commence une nouvelle transaction dans l'ATS en utilisant l'adaptateur JTS. Une transaction adaptée est alors créée et activée par l'ATS. Le contexte transactionnel associe dès lors tout appel de méthode de l'application tierce avec cette nouvelle transaction.
2. L'application tierce interroge l'application de réservation de vols en utilisant les moyens d'invocation offerts par la plate-forme CORBA sur laquelle s'exécute l'application. Le contexte transactionnel est alors propagé de l'application tierce vers l'application de réservation des vols en utilisant le mécanisme d'interception CORBA (en anglais, *CORBA Portable Interceptors*) [WPSO01]. En particulier, l'intercepteur côté client définit l'adaptateur OTS de l'ATS comme le service de transactions courant à utiliser. Ensuite, l'intercepteur côté serveur lit cette information à la réception d'une requête d'invocation de méthode et il associe cet appel avec l'ATS. Par conséquent, les objets de type `Resource` et `Synchronization` de l'application de réservation des vols sont alors associés à la transaction gérée par l'ATS.
3. L'application tierce interroge ensuite le service de réservation d'hôtels via les mécanismes offerts par la plate-forme *Web Services*. Le contexte transactionnel est alors propagé de l'application tierce vers l'application de réservation d'hôtels en utilisant un contexte de coordination (en anglais, *WS-Coordination context*) [CCF+05b] dans l'entête de la requête SOAP (en anglais, *Simple Object Access Protocol*) émise pour interroger le *Web Service*. Cela signifie que le *Web Service* de réservation d'hôtels va utiliser l'adaptateur WS-AT de l'ATS pour associer à la transaction courante tout participant de type `TransactionAwareParticipant`.
4. Finalement, l'application valide la transaction qu'elle avait activée. Cette opération a pour conséquence l'exécution du protocole de validation en deux phases configuré dans le CTS

sur tous les participants de la transaction quel que soit leur type. Ainsi, les participants des trois plates-formes applicatives collaborent pour déterminer l'issue de la transaction et applique cette issue de façon uniforme.

10.4 Mise en œuvre et évaluation du service transactions adapté

Cette section présente la réalisation du service de transactions adapté présenté dans la section précédente en utilisant le canevas logiciel GoTM. Les performances de cette implémentation sont ensuite comparées à un service de transactions existant.

10.4.1 Éléments d'implantation avec GoTM

L'implantation de l'ATS avec GoTM requiert de mettre en œuvre les stratégies, les adaptateurs et le bus logiciel en utilisant les composants disponibles dans le canevas logiciel GoTM. Dans GoTM, tout service de transactions se compose essentiellement de deux parties : la partie statique et la partie dynamique comme nous l'avons détaillé dans le chapitre 7.

La partie statique, illustrée dans la figure 10.10, regroupe les fonctionnalités de gestion des instances de transactions en cours d'exécution (composants Transaction Current et Transaction Active) et de création de nouvelles transactions (composants Transaction Factory, Transaction Model et Transaction Created). Ces fonctionnalités différent peu dans les trois standards transactionnels que nous considérons. Néanmoins, il est nécessaire de placer des adaptateurs devant ces fonctionnalités afin de garantir la compatibilité avec les interfaces des standards. Ces adaptateurs partagent les fonctionnalités génériques (composant Common Transaction Service) en les adaptant aux interfaces des standards transactionnels.

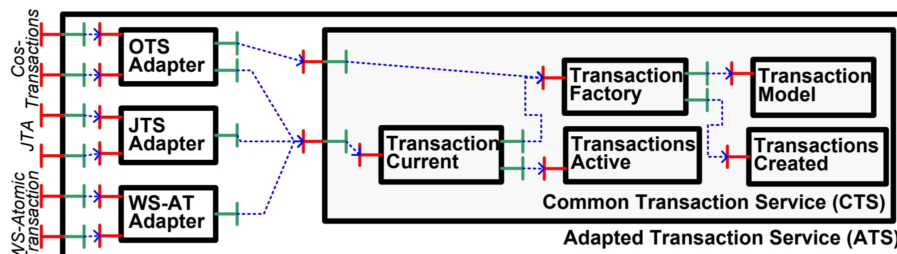


FIG. 10.10: Description de la partie statique du service de transactions adapté.

La partie dynamique du service de transactions, illustrée dans la figure 10.11, représente les instances de transactions qui sont créées par le service. Ces transactions sont associées à un comportement défini par les composants décrivant la transaction.

Dans le cadre du service de transactions adapté, ces composants sont les stratégies, les adaptateurs et le bus logiciel que nous avons définis dans la section précédente. Le bus de messages est ainsi réalisé avec le composant Message Bus (présenté dans la section 7.5.7). La stratégie *2PC* est réalisée avec le composant 2PC Protocol (présenté dans la section 7.5.4). La stratégie *AtomicTransactionState* est réalisée avec le composant présenté dans la section 7.5.5. Ces trois premiers éléments (composant Common Transaction) sont communs aux trois standards transactionnels et sont, par conséquent, partagés entre les adaptateurs OTS Adapter, JTS Adapter et WS-AT Adapter. Chacun de ces adaptateurs est ensuite associé à un gestionaire de participants qui lui est propre. Ainsi, les stratégies *XAResources* et *SynchronizationsJTA* sont réalisées avec le composant JTA Participant Manager (présenté dans la section 7.5.6). Les stratégies *Resources*, *Synchronizations* et *SubTransactionAwareResources* sont réalisées avec le composant OTS Participant Manager. La stratégie *TransactionAwareParticipants* est réalisée avec le composant WS-AT Participant Manager. Les composants JTS Status Converter, OTS Status Converter et WS-AT Status Converter

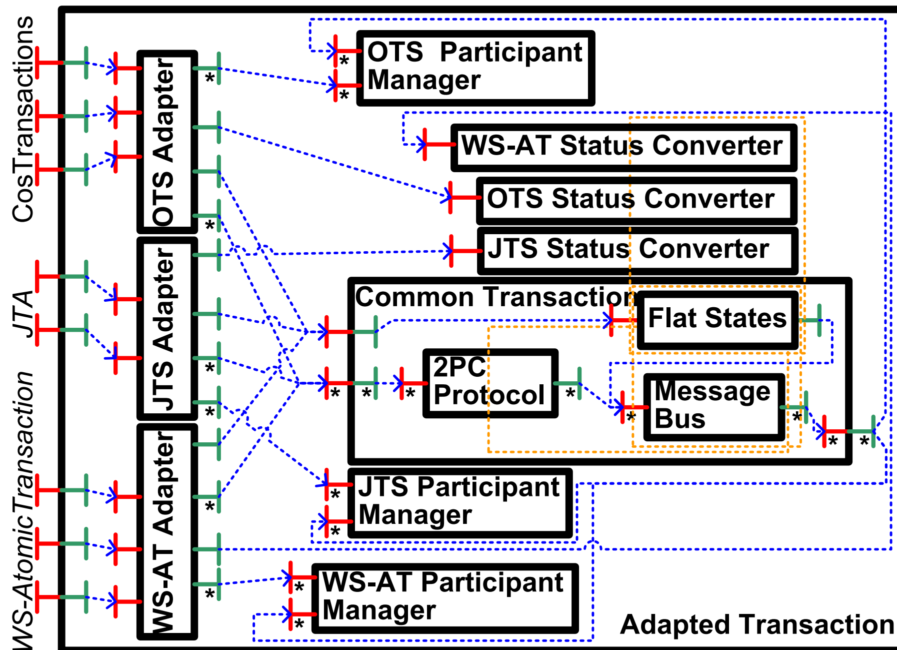


FIG. 10.11: Description de la partie dynamique du service de transactions adapté.

permettent de convertir l'état de la transaction dans l'un des formalismes utilisés respectivement par les standards JTS, OTS et WS-AT.

10.4.2 Évaluation des performances

Dans cette section, nous confrontons notre implantation du service de transactions adapté à un service de transactions existant. L'objectif de ce comparatif est de montrer que (1) l'utilisation d'un modèle à composants pour construire des services de transactions est pertinent et (2) le modèle de service présenté dans ce chapitre supporte le passage à l'échelle en terme de participants et de transactions concurrentes.

Nous comparons notre implantation de ATS au service de transactions JOTM [Mes03] développé dans le cadre du consortium ObjectWeb. JOTM est une implantation en Java du standard transactionnel JTS [Che99] reconnue pour sa fiabilité et ses performances. JOTM est le moteur de transactions utilisé par le serveur d'applications J2EE JONAS [Exe04]. Nous utilisons la version 1.5.10 de JOTM afin de comparer les services à fonctionnalités équivalentes. Notre implantation d'ATS utilise AOKELL [SPDC06] comme substrat d'exécution du modèle de composants FRACTAL. Nous utilisons un Intel Pentium4 2 GHz avec 1024 Mo de RAM (DELL Optiflex GX 240) comme plate-forme d'exécution. L'expérience est réalisée sur une seule machine afin d'éviter le bruit engendré par le trafic réseau. Le système d'exploitation utilisé est un Linux Ubuntu basé sur la version $2.6.10 - 386$ du noyau et nous utilisons le JDK 1.5.0_04 comme implantation de Java.

La figure 10.12 évalue le passage à l'échelle des services de transactions ATS et JOTM en observant le temps d'exécution d'une transaction (en millisecondes) en fonction de son nombre de participants. Cette évaluation est réalisée en utilisant une application exécutant séquentiellement plusieurs transactions en faisant varier le nombre de participants de 0 à 50. Chaque mesure est basée sur une moyenne de 1000 exécutions.

Cette évaluation empirique montre que l'ATS fournit de meilleurs temps d'exécution que le service de transactions JOTM lorsque le nombre de participants d'une transaction varie. Nous montrons ainsi que l'utilisation d'une approche à base de composants à grains fins n'entraîne pas

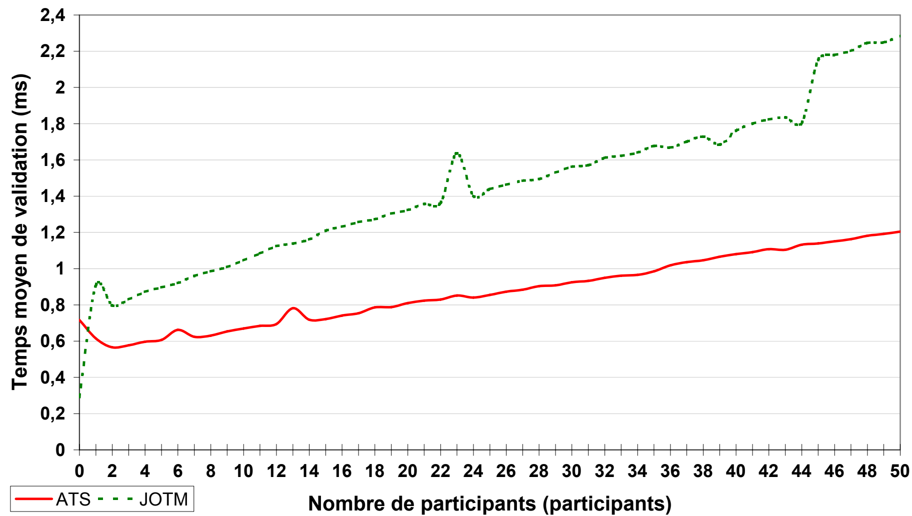


FIG. 10.12: Passage à l'échelle du nombre de participants.

de surcoût à l'exécution comparé à une implantation orientée objet d'une même fonctionnalité. Bien au contraire, les transactions supportées par l'ATS s'exécutent en moyenne 1,5 fois plus vite que les transactions supportées par le service JOTM.

La figure 10.13 évalue le passage à l'échelle des services de transactions ATS et JOTM en observant le nombre de transactions supportées par seconde en fonction du nombre de transactions concurrentes en cours d'exécution. Cette évaluation est réalisée en utilisant une application cliente *multi-thread* exécutant de 1 à 51 transactions en parallèle. Chaque transaction comporte 40 participants. Chaque mesure est basée sur une moyenne de 1000 exécutions.

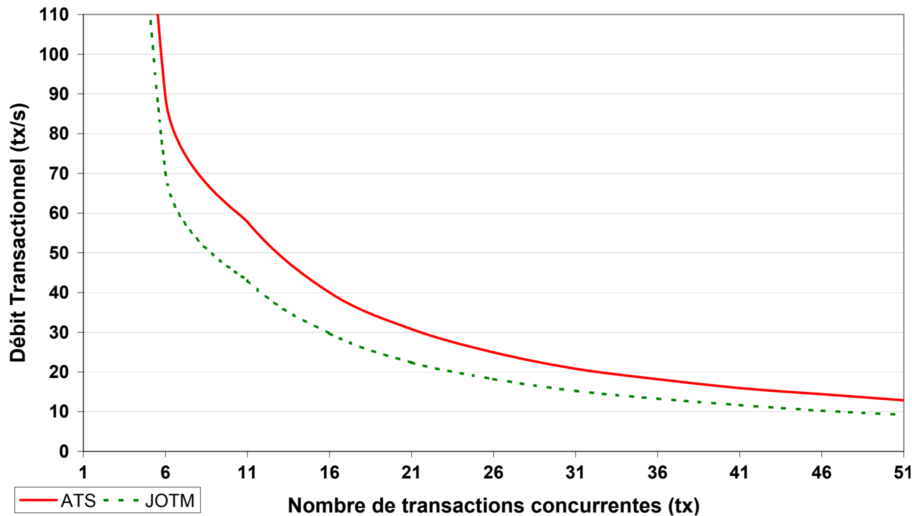


FIG. 10.13: Passage à l'échelle du nombre de transactions.

Cette évaluation empirique montre que l'architecture de l'ATS supporte le passage à l'échelle lorsque le nombre de requêtes reçues par les applications augmente. Ce résultat est obtenu grâce à la délégation du comportement transactionnel à la transaction (par exemple, le protocole de coordination). Cette isolation permet à plusieurs transactions d'être validées simultanément.

10.5 Travaux connexes

Dans cette section, nous comparons le service de transactions adapté à d'autres approches adressant la synchronisation ou l'interopérabilité de transactions.

10.5.1 Interopérabilité des standards transactionnels

Le standard transactionnel *Java Transaction Service* (JTS) [Che99] propose une solution pratique afin de répondre aux problèmes d'interopérabilité et de composition des transactions. En particulier, JTS repose sur le standard transactionnel *Object Transaction Service* (OTS) [OMG03] afin de propager les contextes transactionnels entre les services du type JTS. Ce choix permet en théorie de faire coopérer des services de transactions basés sur les standards JTS et OTS. Cependant, à l'heure actuelle, seule la suite de services de transactions Arjuna [Lit05] supporte cette architecture. De plus, il s'avère qu'une telle réutilisation des protocoles n'est pas toujours aussi simple. Par exemple, les standards *Web Services Atomic Transaction* (WS-AT) [CCF+05a] et *Web Services Business Activity* (WS-BA) [CCF+05c] utilise l'entête des requêtes SOAP pour propager le contexte transactionnel. Notre approche fait abstraction des standards transactionnels existants pour éviter ce type de dépendance. Par conséquent, les transactions peuvent être propagées indépendamment des standards supportés par l'ATS. La propagation des contextes transactionnels est réalisée par les mécanismes adaptés pour chaque type de plate-forme distribuée (par exemple, *CORBA Portable Interceptors* [WPSO01], *WS-Coordination* [CCF+05b]).

Ces dernières années, le standard *Web Services Atomic Transaction* (WS-AT) [CCF+05a] a proposé une abstraction du service de transactions afin de fournir des propriétés ACID lors de la coordination de l'exécution de plusieurs *web services*. Dans cette architecture, tout *web service* souhaitant participer à la transaction s'enregistre auprès d'un coordinateur global supportant le standard WS-AT. Les *web services* impliqués dans la transaction sont alors synchronisés à la fin de la transaction par un protocole de validation (par exemple, protocole de validation en deux phases). Cette solution homogène pour l'interopérabilité de systèmes patrimoniaux semble séduisante mais requiert que toute application souhaitant être synchronisée avec d'autres applications soit disponible sous la forme d'un *web service*. Les interactions entre les applications doivent également être réalisées via la plate-forme *Web Services* et le protocole de communication SOAP. Or, il s'avère que les performances du protocole SOAP restent bien inférieures aux performances d'autres protocoles comme *Internet Inter-ORB Protocol* (IIOP) ou *Remote Method Invocation* (RMI) [DHRM05].

10.5.2 Adaptation des services de transactions

Finalement, d'autres approches à base de composants, abordées dans le chapitre 2, ont proposé de construire des services de transactions adaptables [HNL04, AK05]. Dans ces approches de plus grosse granularité, les services de transactions sont considérés comme des composants monolithiques dont on souhaite faciliter l'intégration dans une application. Dès lors, différents services de transactions sont mis à disposition de l'application en utilisant un mécanisme de courtage (en anglais, *Trading Service*). L'inconvénient de ces approches réside dans l'explosion exponentielle que représentent toutes les possibilités de compositions de standards. En plus des différentes propriétés non-fonctionnelles que peuvent supporter les services de transactions (par exemple, protocole de validation, modèle de transaction), il peut s'avérer difficile de trouver un service de transactions en adéquation avec les besoins de l'application. Notre approche repose sur l'utilisation de composants de fine granularité. Ce choix nous permet de réutiliser plus facilement les différents éléments d'un service de transactions afin de construire un ATS répondant aux critères requis par une application.

10.6 Conclusion

Ce chapitre a présenté une approche pour la construction de services de transactions adaptés aux différents standards transactionnels disponibles sur le marché. La conception d'un service de transactions adapté (en anglais, *ATS pour Adapted Transaction Service*) est guidée par l'analyse des standards transactionnels que le service doit supporter. L'analyse permet ainsi d'identifier les différentes fonctions requises par le service de transactions et les stratégies qui doivent être associées à ces services. L'ATS résultant est alors une composition des stratégies requises et des adaptateurs (chaque adaptateur assure la compatibilité avec un standard transactionnel particulier) en utilisant un bus logiciel afin de faciliter les échanges de messages entre les différentes stratégies. L'ATS peut ensuite être implanté avec le canevas logiciel GOTM comme présenté dans ce chapitre. Nous montrons qu'une telle implantation propose des performances acceptables en la comparant au service de transactions JOTM.

Notre proposition facilite donc la composition de standards transactionnels car (1) l'ATS est utilisé de manière transparente par l'application et (2) l'ATS n'affecte ni la complexité des plates-formes existantes ni les performances de celles-ci.

Les perspectives relatives à ces travaux adressent la prise en compte de nouveaux standards transactionnels. En effet, nous pensons que notre approche peut facilement supporter d'autres types de standards transactionnels tels que les modèles de transactions étendues [CCF+05c, OMG05a].

Les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Patricia Serrano-Alvarado et ils ont donné lieu à deux publications internationales [RM04b, RSAM06a].

A program that compiles on the first run has an error in the algorithm.

Murphy's Law

Chapitre 11

CATE : l'adaptation du protocole de validation lors de l'exécution

Sommaire

11.1 Introduction **196**

11.2 Présentation des protocoles de validation en deux phases **196**

11.2.1 Protocole *Two-Phase Commit* (2PC) 197

11.2.2 Protocole *Two-Phase Commit Presumed Abort* (2PC-PA) 197

11.2.3 Protocole *Two-Phase Commit Presumed Commit* (2PC-PC) 198

11.3 Évaluation des protocoles de validation en deux phases **199**

11.3.1 Évaluation théorique 199

11.3.2 Élements d'implantation 199

11.3.3 Évaluation empirique 201

11.3.4 Discussions 203

11.4 CATE : un service de transactions sensible au contexte **203**

11.4.1 Sensibilité au contexte 204

11.4.2 Règles de reconfiguration 205

11.4.3 Politique d'adaptation du protocoles de validation en deux phases 205

11.4.4 Évaluation empirique des performances de CATE 206

11.4.5 Discussions 207

11.5 Travaux connexes **207**

11.6 Conclusion **208**

DEPUIS PLUSIEURS ANNÉES, de nombreux protocoles transactionnels ont été définis pour répondre à des besoins applicatifs particuliers. Traditionnellement, lors de l'implantation d'un service de transactions, un protocole particulier est choisi et demeure le même durant toute l'exécution du système. Cependant, l'aspect dynamique des contextes applicatifs actuels (par exemple, informatique ubiquitaire, pair-à-pair, grille) et les différentes variations contextuelles possibles (principalement dûes à la sémantique des applications) nécessite désormais de prendre en compte l'adaptation des services de transactions. C'est pourquoi la prochaine génération de services de transactions doit être adaptable et même auto-adaptable.
*Ce chapitre présente CATE (*Context-Aware Transaction Service*) qui est (1) une architecture à base de composants des protocoles de validation à deux phases standards et (2) un service de transactions sensible au contexte. La construction du service de transactions sensible au contexte et des différentes versions des protocoles de validation s'appuient sur le canevas* GOTM*. L'auto-adaptation dans CATE est réalisée par observation du contexte et reconfiguration de l'architecture à base de composants. CATE choisit ainsi le protocole de validation le plus approprié au contexte d'exécution courant. Nous montrons que l'utilisation de CATE fournit de meilleurs performances que les services de transactions basés sur un seul protocole lorsque le contexte d'exécution de l'application est soumis à de nombreuses variations.*

11.1 Introduction

Dans les systèmes distribués, les protocoles de validation assurent la propriété d'atomicité, comme nous l'avons introduit dans le chapitre 2. Cette propriété garantit que, dans le cadre d'une transaction, toutes les opérations doivent être validées ensemble ou toutes abandonnées. Le protocole de validation le plus utilisé est le protocole de validation en deux phases (2PC) [Gra78]. Ce protocole a fait l'objet de nombreuses optimisations et certaines d'entre elles sont tellement utilisées qu'elles sont intégrées dans les standards. Ces différentes variations des protocoles de validation sont proposées afin de réduire le coût des transactions exécutées, d'adresser une sémantique particulière (par exemple, les transactions en lecture seule), de s'exécuter sur différentes topologies de réseaux, etc. Par exemple, le protocole *2PC Presumed Commit* (2PC-PC) [MLO86] est particulièrement adapté aux transactions présentant un taux de validation important. À l'inverse, le protocole *2PC Presumed Abort* (2PC-PA) [MLO86] s'attache à réduire le coût d'exécution des transactions qui abandonnent souvent.

En pratique, les services de transactions font le choix d'un protocole de validation particulier. Ce protocole ne change pas durant l'exécution du service de transactions quel que soit le contexte d'exécution. Cette dépendance forte du protocole de validation au service de transactions peut entraîner une dégradation des performances du service dans certains contextes. Pour adresser ce problème de variation du contexte d'exécution et de son impact sur les services de transactions, nous proposons d'introduire de l'auto-adaptabilité dans les services de transactions. Cette auto-adaptabilité doit permettre d'utiliser le protocole de validation le plus adapté au contexte d'exécution courant.

Après avoir introduit les protocoles de validation 2PC, 2PC-PA et 2PC-PC, nous présentons dans ce chapitre CATE. CATE se compose de (1) une architecture à base de composants des protocoles de validation standards et (2) un service de transactions sensible au contexte. Nous montrons finalement que les transactions supportées par CATE s'exécutent plus vite que les transactions supportées par les services de transactions utilisant un seul protocole de validation.

11.2 Présentation des protocoles de validation en deux phases

Dans cette section, nous nous intéressons aux protocoles de validation en deux phases du type 2PC, 2PC-PA et 2PC-PC. La description comportementale de ces protocoles est réalisée en

utilisant le diagramme UML de séquences que nous avons introduit dans le chapitre 8. Nous utilisons quatre acteurs pour modéliser le protocole : Application, Cooordinator, Participants et Log. Les séquences décrivent le comportement des protocoles 2PC, 2PC-PA et 2PC-PC en terme d'échanges de messages et d'écritures sur le disque. Ces écritures sur le disque permettent de journaliser la progression du protocole de validation aussi bien du côté du coordinateur que des participants. Le journal résultant est utilisé en cas d'arrêt suspect du service de transactions afin de reprendre éventuellement les transactions qui n'étaient pas complétées. Il existe deux types d'écritures : l'écriture forcée et l'écriture non forcée. La première écrit immédiatement les données en mémoire persistante en générant un accès disque. La seconde mémorise l'écriture dans un tampon qui est transféré sur le disque périodiquement. Par conséquent, l'utilisation des écritures non forcées introduit une fenêtre de vulnérabilité dans les protocoles de validation tant que ces écritures ne sont pas transférées définitivement sur le disque. Le cas d'utilisation *Abort* considère les transactions abandonnant de façon unilatérale — c.-à-d. sur ordre de l'application — tandis que le cas *Failure* représente un abandon de la transaction décidé à l'issue de la phase de vote du protocole de validation.

11.2.1 Protocole *Two-Phase Commit* (2PC)

Le protocole 2PC consiste en deux phases comme illustré dans la figure 11.1. Durant la phase de vote, le coordinateur envoie un message *prepare* à tous les participants. Durant la phase de décision, le coordinateur collecte les réponses des participants, puis il décide de l'issue de la transaction. Cette issue peut être la validation (*commit*) si tous les participants ont voté *oui* ou l'abandon (*abort*) si au moins l'un des participants a voté *non*. Lorsque l'issue est déterminée par le coordinateur, celui-ci envoie un message à tous les participants contenant l'issue qu'il a choisi. Lorsque les participants reçoivent cette décision, ils confirment ou infirment leurs modifications avant d'envoyer un accusé de réception (*acknowledge*) au coordinateur. Le protocole de validation se termine quand le coordinateur reçoit tous les accusés de réception des participants qui avaient voté *oui* durant la phase de décision.

Pour le protocole 2PC classique, le coordinateur force l'écriture de la décision prise quant à l'issue de la transaction. Il utilise une écriture non forcée pour mémoriser la fin du protocole. Les participants forcent l'écriture de leur vote ainsi que la décision qu'ils reçoivent du coordinateur. Ces opérations d'écriture sont réalisées avant l'envoi des messages correspondants.

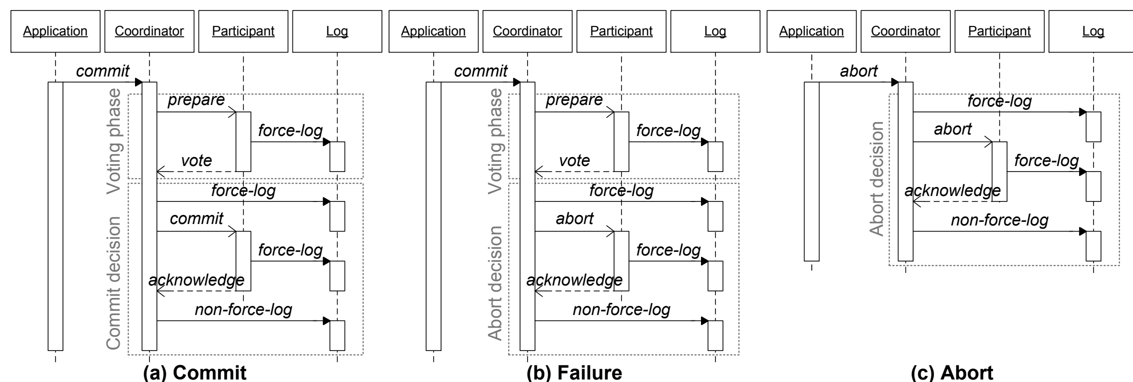


FIG. 11.1: Le protocole *Two-Phase Commit* (2PC).

11.2.2 Protocole *Two-Phase Commit Presumed Abort* (2PC-PA)

Le protocole 2PC-PA réduit le coût associé aux transactions présentant un taux élevé d'abandons (voir figure 11.2). Ainsi, lorsque le coordinateur décide d'abandonner la transaction, il élimine toutes les informations relatives à la transaction et il envoie un message *abort* à tous

les participants sans mémoriser la décision (voir figure 11.2(b)). Les participants utilisent une écriture non forcée pour mémoriser le message *abort* et ne sont pas tenus d'envoyer un accusé de réception au coordinateur.

Toute information manquante dans le journal est donc interprêtée comme un abandon de la transaction. Le comportement de la validation d'une transaction utilisant le protocole 2PC-PA est en tout point identique à celui des transactions utilisant le protocole 2PC (voir figure 11.2(a)).

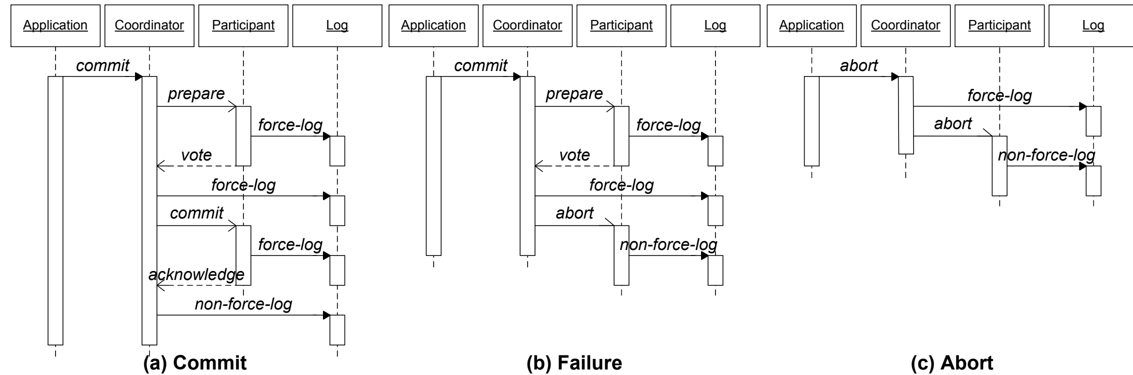


FIG. 11.2: Le protocole *Two-Phase Commit Presumed Abort* (2PC-PA).

11.2.3 Protocole *Two-Phase Commit Presumed Commit* (2PC-PC)

Par opposition au protocole de validation 2PC-PA, le protocole 2PC-PC s'intéresse à l'optimisation des transactions qui présentent un fort taux de validation. Toute information manquante dans le journal est interprêtée ici comme une validation de la transaction. Pour cela, le coordinateur doit forcer l'écriture d'un message *initiation* dans le journal avant l'envoi du message *prepare* aux participants (voir figure 11.3). Lorsqu'il décide de valider la transaction, le coordinateur force l'écriture de la décision *commit* avant d'envoyer le message associé à tous les participants (voir figure 11.3(a)). De leur côté, les participants utilisent une écriture non forcée pour mémoriser cette décision, libèrent leurs ressources sans envoyer de messages *acknowledge* au coordinateur. Dans le cas d'un abandon, la décision n'est pas mémorisée et le message *abort* est envoyé aux participants qui ont voté *oui* et le coordinateur se met en attente des accusés de réception (voir figure 11.3(b)). Les participants forcent alors l'écriture de la décision. Lorsque tous les accusés de réception ont été reçus, le coordinateur utilise une écriture non forcée pour marquer la fin du protocole et il élimine toutes les informations relatives à la transaction.

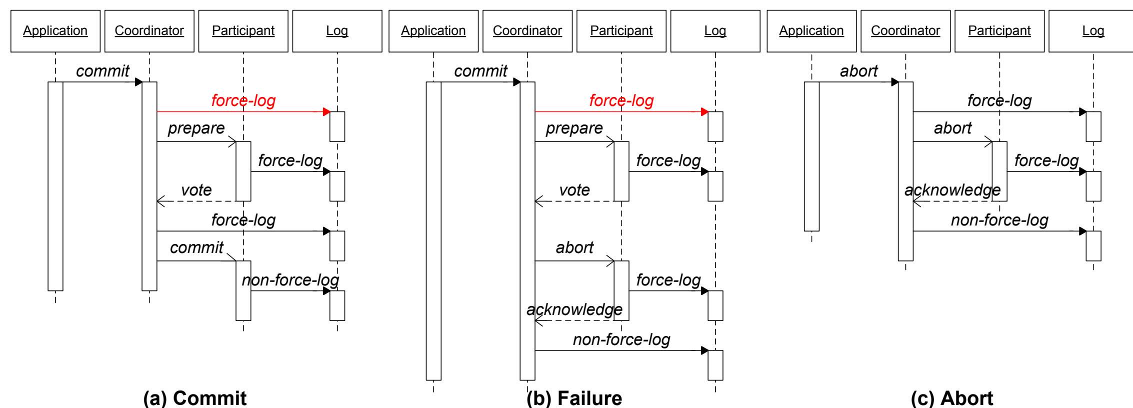


FIG. 11.3: Le protocole *Two-Phase Commit Presumed Commit* (2PC-PC).

11.3 Évaluation des protocoles de validation en deux phases

L'évaluation des protocoles de validation est souvent basée sur une évaluation théorique du coût en fonction du nombre de messages échangés entre le coordinateur et les participants ainsi qu'en terme de nombre d'écritures forcées réalisées par le coordinateur et les participants. Dans cette section, nous souhaitons valider les mesures théoriques réalisées dans la littérature par un ensemble de mesures empiriques. Ces mesures empiriques sont réalisées sur une implantation des protocoles de validation utilisant la bibliothèque GOTM. L'originalité de cette implantation réside dans la définition de composants réutilisables quel que soit le protocole considéré.

11.3.1 Évaluation théorique

Les protocoles 2PC, 2PC-PA et 2PC-PC diffèrent par le nombre de messages qu'ils échangent et le nombre d'écritures forcées qui sont réalisées. Le tableau 11.1 résume les coûts de chaque protocole. Les différences entre les protocoles impactent non seulement le temps d'exécution du protocole de validation mais également le trafic des messages sur le réseau et l'espace consommé sur les disques par les journaux. Le cas d'utilisation *Abort* considère les transactions abandonnant de façon unilatérale — c.-à-d. sur ordre de l'application — tandis que le cas *Failure* représente un abandon de la transaction décidé à l'issue de la phase de vote du protocole de validation.

Commit protocol	Messages			Écritures forcées		
	Commit	**Failure**	**Abort**	**Commit**	**Failure**	**Abort**
2PC	$4p$		$2p$	$1+2p$		$1+p$
2PC-PA	$4p$	$3p$	p	$1+2p$	$1+p$	1
2PC-PC	$3p$	$4p$	$2p$	$2+p$	$1+2p$	$1+p$

TAB. 11.1: Coût théorique des protocoles 2PC.

Si le protocole de validation 2PC traditionnel apparaît comme le plus utilisé, celui-ci demeure néanmoins le plus couteux. En effet, le protocole 2PC échange $4p$ messages (p étant le nombre de participants) et il force l'écriture de $1 + 2p$ enregistrements (le coût des écritures non forcées est volontairement négligé). Ce coût du protocole 2PC a été à l'origine de nombreuses optimisations.

À l'économie d'une écriture forcée du côté du coordinateur et de chaque participant, le protocole 2PC-PA ajoute l'économie du message *acknowledge* émis par chaque participant dans le cadre d'une transaction abandonnée. Par conséquent, le coût d'un abandon d'une transaction est évalué à $3p$ messages et p écritures forcées. Si la transaction abandonne de façon unilatérale, le coût du protocole n'est que de 1 message et 1 écriture forcée, ce qui le rend beaucoup plus performant que les protocoles 2PC et 2PC-PC. Le coût d'une transaction validée est identique au coût d'une transaction utilisant 2PC.

Comparé à 2PC, 2PC-PC économise une écriture forcée et un message *acknowledge* pour chaque participant dans le cas d'une transaction validée. Le protocole 2PC-PC diffère du protocole 2PC-PA par l'introduction d'une écriture forcée *initiation* au niveau du coordinateur. En conséquence, le coût du protocole de validation 2PC-PC est évalué à $2 + p$ écritures forcées et $3p$ messages pour les transactions validées. Dans le cas d'un abandon de la transaction, le coût du protocole est égal au coût du protocole 2PC majoré par l'écriture forcée *initiation*.

Le protocole 2PC-PA est adapté à un système dont les transactions abandonnent en majorité tandis que le protocole 2PC-PC est plus adapté si les transactions ont tendance à être validées. Notons que dans un système dont les transactions ont la même probabilité de valider ou d'abandonner, le protocole 2PC-PA reste plus adapté que le protocole 2PC-PC.

11.3.2 Élements d'implantation

Cette section présente une implantation possible des protocoles de validation 2PC, 2PC-PA et 2PC-PC en utilisant une architecture à base de composants. L'architecture que nous propo-

sons repose sur le modèle de composants FRACTAL et généralise les protocoles de validation afin de réutiliser au maximum les fonctionnalités communes. Ces fonctionnalités communes sont fournies par le canevas logiciel GOTM. Notre objectif est (1) de réaliser une implantation orientée composants des trois protocoles de validation que nous considérons et (2) d'exprimer les différentes sémantiques des protocoles au travers des liaisons qui unissent ces composants (voir la figure 11.4).

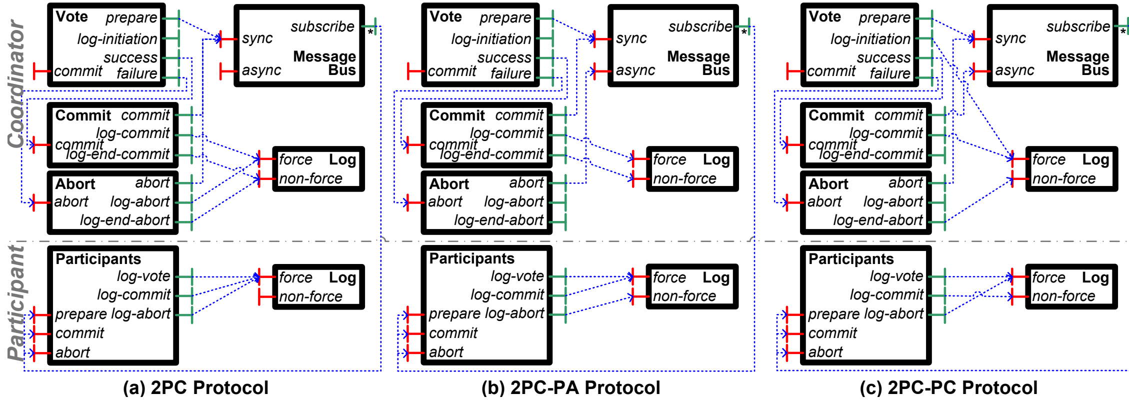


FIG. 11.4: Implantation à base de composants des protocoles 2PC.

L'acteur Application présent dans les diagrammes UML de séquences (voir figures 11.1 à 11.3) ne nécessite pas d'être représenté au niveau de l'implantation du protocole de validation. Dans un premier temps, nous définissons trois composants : Coordinator, Participants et Log. Ensuite, afin de faciliter la réutilisation des composants, le composant Coordinator est divisé en trois sous-composants : Vote, Commit et Abort. Cette séparation permet non seulement de réifier à l'exécution les phases caractéristiques du protocole de validation mais il met également en avant le composant dédié à l'abandon d'une transaction. Ce composant est réutilisé pour réaliser l'abandon unilatéral des transactions. La communication est supportée par un composant Message Bus. Ce composant permet l'envoi de messages synchrones ou asynchrones (via un mécanisme de *callback*) que nous avons présenté dans le chapitre 7.

Dans les figures 11.1 à 11.3, le coordinateur émet les messages *prepare*, *commit* et *abort* à destination de tous les participants. Par conséquent, les interfaces clientes *prepare*, *commit* et *abort* des composants du coordinateur sont liées aux interfaces serveur *sync* et *async* du composant Message Bus selon le type des messages échangés entre le coordinateur et les participants dans le diagramme UML de séquences (comme présenté dans le chapitre 8).

De la même façon, les composants du coordinateur et les participants requièrent l'accès à un journal pour mémoriser l'évolution du protocole de validation. Par conséquent, les composants du coordinateur et les participants sont connectés aux interfaces serveur *force* et *non-force* fournies par le composant Log (introduit dans le chapitre 4) selon la configuration exprimée dans le diagramme UML de séquences.

Les composants du coordinateur sont inclus dans le service de transactions tandis que les participants sont généralement pris en charge par des gestionnaires de ressources (par exemple, des gestionnaires de base de données) supportés par le système. Notre implantation des trois protocoles de validation réutilise les six composants évoqués précédemment. Chaque implantation de protocole 2PC diffère uniquement par les liaisons qui assemblent ces six composants.

Le protocole 2PC (figure 11.4(a)). Le coordinateur émet un message *prepare* synchrone destiné à tous les participants. Chaque participant attache son *vote* au message *callback* renvoyé au coordinateur. Lorsqu'une décision est prise, le coordinateur mémorise l'issue de la transaction en utilisant son interface cliente *log-commit* (ou *log-abort*), celle-ci est connectée à l'interface *force* du composant Log. Le message *commit* (ou *abort*) est envoyé de manière synchrone afin que les participants puissent répondre par un message *acknowledge*. Le co-

ordinateur marque la fin du protocole par l'écriture non-forcée d'un enregistrement *end* en utilisant l'interface client *log-end-commit* (ou *log-end-abort*). Le composant Participants reçoit les messages *prepare*, *commit*, et *abort* via le composant Message Bus. Les décisions reçues par les participants sont systématiquement écrites de force dans le journal.

Le protocole 2PC-PA (figure 11.4(b)). L'interface cliente *log-abort* n'est liée à aucune interface serveur étant donné que le coordinateur ne mémorise par les décisions du type *abort*. Le message *abort* est quant à lui envoyé de façon asynchrone aux participants étant donné qu'aucun accusé de réception n'est requis. Finalement, l'interface cliente *log-end-abort* n'est pas connectée car la fin d'une transaction abandonnée n'est pas mémorisée dans le protocole 2PC-PA. L'interface cliente *log-abort* du composant Participants est connectée à l'interface *non-force* du composant Log. Par conséquent, le code des composants du protocole sont identiques, seules les liaisons entre ces composants nécessitent d'être redéfinies dans le cas de l'abandon d'une transaction. En effet, l'implantation du cas où les transactions valident est identique à l'implantation fournie par le protocole 2PC.

Le protocole 2PC-PC (figure 11.4(c)). L'envoi du message *prepare* est précédé d'une écriture forcée d'un enregistrement via l'interface *log-initiation*. Contrairement aux autres protocoles, cette interface est connectée à l'interface *force* du composant Log. La décision *commit* est envoyée via un message asynchrone et l'interface *log-end-commit* n'est pas connectée. La décision *abort* est envoyée via un message synchrone mais elle n'est pas mémorisée. La fin d'une transaction abandonnée est mémorisée via une écriture non-forcée. Les interfaces clientes *log-commit* et *log-abort* du composant Participants sont connectées respectivement aux interfaces serveurs *non-force* et *force* du composant Log.

11.3.3 Évaluation empirique

L'objectif de cette section est de valider les mesures théoriques présentées dans la section 11.3.1. Nous proposons ainsi d'utiliser l'implantation orientée composants des protocoles 2PC présentés dans la section 11.3.2 et d'observer leurs performances en conditions simulées d'utilisation. Cette comparaison doit (1) valider nos implantations des protocoles de type 2PC et (2) confirmer les conclusions faites à la vue des évaluations théoriques.

Le scénario présenté dans les figures 11.5 à 11.7 évalue le temps moyen d'exécution d'une transaction en fonction du nombre de participants qui lui sont associés (de 0 à 5 participants). Le temps moyen d'exécution est basé sur une moyenne de 1000 exécutions. Notre implantation des protocoles de type 2PC utilise AOKell [SPDC06] comme substrat d'exécution du modèle de composants FRACTAL. Nous utilisons un Intel Pentium4 2 GHz avec 1024 Mo de RAM (DELL Optiflex GX 240) comme plate-forme d'exécution. L'expérience est réalisée sur une seule machine afin d'éviter le bruit engendré par le trafic réseau. Le système d'exploitation utilisé est un Linux Ubuntu basé sur la version $2.6.10-386$ du noyau et nous utilisons le JDK $1.5.0_04$ comme implantation de Java.

Dans la figure 11.5, toutes les transactions exécutées sont validées. Dans cette situation, le protocole 2PC-PC est plus performant que les protocoles 2PC et 2PC-PA pour les transactions impliquant au moins 1 participant. Le surcoût observé pour les transactions n'impliquant pas de participants est dû à l'écriture forcée introduite à l'initiation du protocole 2PC-PC. Les performances de 2PC et 2PC-PA sont équivalentes car leurs implantations du cas *commit* sont identiques.

Dans la figure 11.6, toutes les transactions abandonnent durant l'exécution du protocole de validation. Dans ce cas, le protocole 2PC-PA présente des performances meilleures que les protocoles 2PC et 2PC-PC. Ce gain est expliqué par l'économie du message *acknowledge* et des écritures forcées réalisées dans cette version du protocole de validation. Les performances de 2PC et 2PC-PC sont équivalentes car leurs implantations du cas *abort* sont très proches (à la différence de l'écriture forcée *initiation*).

Dans la figure 11.7, toutes les transactions sont abandonnées de façon unilatérale et à la demande de l'application. Dans ce cas particulier, le protocole 2PC-PA propose de meilleures

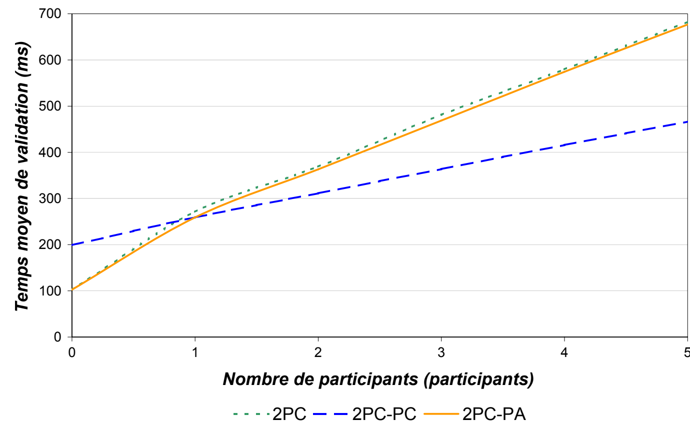


FIG. 11.5: Évaluation des protocoles pour un taux de validation important.

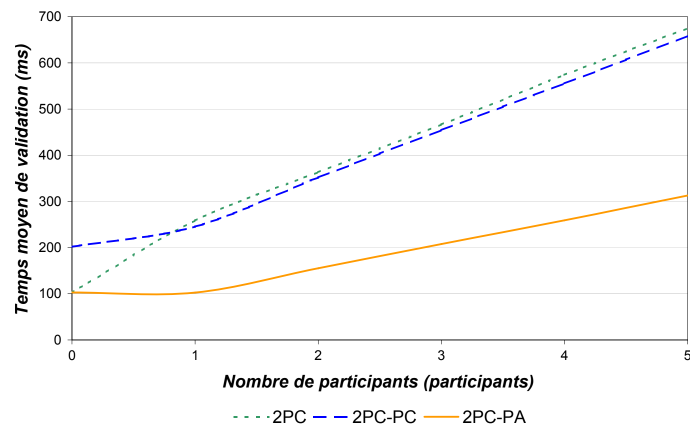


FIG. 11.6: Évaluation des protocoles pour un taux d'abandon important.

performances que les protocoles 2PC et 2PC-PC. En effet, dans le cas d'un abandon unilatéral, le protocole 2PC-PA n'utilise qu'un message asynchrone et qu'une écriture forcée pour signifier l'abandon de la transaction à tous les participants. Ce type d'issue unique pour une transaction est utilisé dans de nombreuses autres optimisations des protocoles de type 2PC [AHCL97, AS02, YWP04] afin d'exploiter les performances de cas particulier d'abandon.

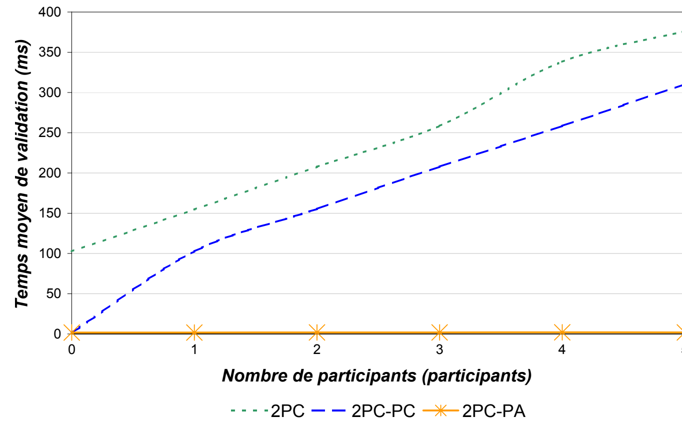


FIG. 11.7: Évaluation des protocoles pour un taux d'abandon unilatéral important.

11.3.4 Discussions

Les mesures réalisées dans la section 11.3.3 ont confirmé la validité des mesures théoriques qui avaient été présentées dans la section 11.3.1. Cette validation a été réalisée en utilisant une implantation des protocoles de type 2PC basés sur les composants du canevas GOTM.

Ainsi, conformément à nos attentes, le protocole 2PC-PC propose les meilleures performances dans un contexte favorisant la validation des transactions alors que le protocole 2PC-PA est bien plus adapté à un contexte favorisant l'abandon des transactions. Cependant, le contexte d'exécution d'une application n'est pas figé et les variations de celui-ci peuvent impacter le taux de validation des transactions et par conséquent les performances du protocole de validation implanté par le service de transactions. À partir de la caractérisation des performances des différents protocoles de type 2PC, il semble pertinent de définir un mécanisme d'adaptation du protocole de validation en fonction du contexte d'exécution.

La prise en compte des transactions abandonnant de façon unilatérale impacte également le choix du protocole de validation. Il apparaît que dans ce cas, le protocole 2PC-PA propose des performances bien meilleures que les autres protocoles. Il semble pertinent d'exploiter les performances du protocole 2PC-PA pour le cas particulier de l'abandon unilatéral des transactions. De plus comme cette situation est initiée à la demande de l'application, il est possible d'appliquer le protocole 2PC-PA systématiquement sans qu'il soit nécessaire d'observer le contexte d'exécution.

11.4 CATE : un service de transactions sensible au contexte

Dans cette section, nous présentons la seconde partie de notre proposition : CATE. CATE est un service de transactions sensible au contexte qui est capable de s'auto-adapter en fonction des variations du contexte d'exécution. L'objectif de CATE est d'utiliser le protocole de validation le plus adapté au contexte d'exécution lorsque l'issue des transactions est imprévisible.

La section 11.4.1 présente le mécanisme de sensibilité au contexte que nous utilisons. La section 11.4.2 présente le mécanisme de reconfiguration appliqué pour sélectionner le protocole de validation le plus adapté. La section 11.4.3 détaille la politique d'auto-adaptation que nous avons définie. La section 11.4.4 compare les performances de CATE à celles des services utilisant un

protocole de validation particulier. Finalement, la section 11.4.5 discute les apports de la solution proposée par CATE.

11.4.1 Sensibilité au contexte

Dans ce chapitre, nous assimilons le taux de validation et d'abandon au contexte d'application. Nous considérons que certaines variations du contexte d'exécution peuvent avoir une influence indirecte sur le taux de validation et d'abandon des transactions. De plus, ce sont ces taux qui déterminent le choix du protocole de validation comme illustré dans la section précédente.

Par conséquent, l'adaptation du protocole de validation au bon moment requiert d'observer le taux de validation et d'abandon des transactions. Le taux de validation dépend de l'occurence du cas d'utilisation *commit* du protocole de validation alors que le taux d'abandon correspond aux cas *failure* et *abort* de ce même protocole. L'algorithme d'adaptation du protocole de validation est appelé *politique d'adaptation* et est défini sous la forme de règles de type Événement/Condition/Action (ECA). L'événement correspond à la variation du taux de validation/abandon, la condition détermine si l'adaptation doit être réalisée alors que l'action effectue concrètement le changement du protocole de validation.

La figure 11.8 présente un service de transactions (composant Tx Manager) et ses liaisons avec les transactions qu'il supporte (composant Tx(2PC-PX)). Le composant Context Awareness implante la politique d'adaptation. Celui-ci observe le nombre de transactions validées et abandonnées et détecte ainsi le changement de contexte, et décide le changement du protocole de validation. Par exemple, la règle ECA rule(abort-rate $<$ 10%, p !=2PC-PC, p=2PC-PC) spécifie que si le taux d'abandon est inférieur à 10% des transactions exécutées, le protocole 2PC-PC doit être utilisé s'il n'était pas déjà configuré.

Afin de comptabiliser le nombre de transactions validées et abandonnées, le composant Context Awareness fournit l'interface *subscribe* requise par le composant Message Bus des transactions (voir la figure 11.4). Le composant Context Awareness peut ainsi être notifié de l'émission des messages *commit* et *abort* par le coordinateur du protocole de validation.

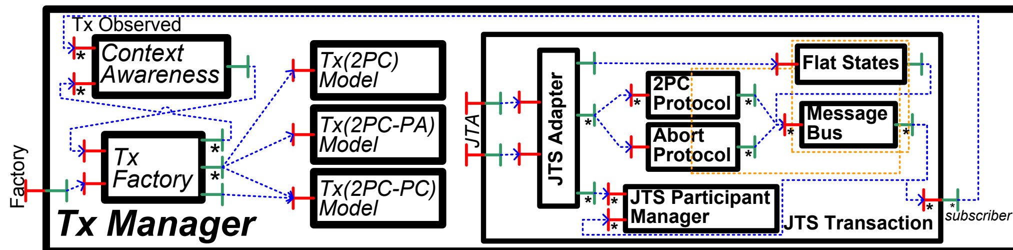


FIG. 11.8: Architecture de CATE.

Le service de transactions illustré dans la figure 11.8 réalise le standard transactionnel JTS [Che99]. Les composants Tx(2PC-Px) représentent des transactions de type JTS implantant le protocole de validation 2PC-Px. Le composant JTS Adapter est connecté aux fonctionnalités fournies par CATE. Ces fonctionnalités incluent les variantes du protocole de validation présenté dans la figure 11.4 et d'autres fonctionnalités plus générales telles que la gestion des états de la transaction. Le composant 2PC Protocol réifie le protocole de validation utilisé par la transaction. Le composant Abort Protocol réifie quant à lui le protocole d'abandon utilisé lors de l'abandon unilatéral des transactions. Ce composant correspond au composant Abort du protocole 2PC-PA présenté dans la figure 11.4. Ce choix permet à CATE de fournir le meilleur temps d'exécution du protocole en cas d'abandon unilatéral de la transaction. Le composant Tx Manager gère les instances des transactions en cours d'exécution. La figure 11.8 représente les liaisons qui connectent le service de transactions aux instances de transactions.

11.4.2 Règles de reconfiguration

Lorsque le composant Context Awareness décide d'effectuer une reconfiguration, celui-ci utilise l'interface *configuration* du composant Tx Factory pour lui spécifier le nouveau protocole de validation à utiliser. Dès lors, le service de transactions utilise le prototype de transactions associé à la configuration courante pour créer les nouvelles instances. La figure 11.8 illustre une configuration où le protocole courant utilisé par le service de transactions est le protocole 2PC-PC.

Lorsque le service de transactions crée une nouvelle transaction, il enregistre l'interface *listener* du composant Context Awareness auprès du composant Message Bus de la transaction afin de pouvoir observer le taux de validation/abandon de celle-ci.

11.4.3 Politique d'adaptation du protocoles de validation en deux phases

La connaissance du taux de validation/abandon courant nous permet d'effectuer une prédiction sur le contexte d'exécution. En particulier, nous considérons que si le taux d'abandon des transactions est d'environ 30% alors cette tendance ne changera pas dans un futur proche.

La condition qui spécifie la nécessité d'une reconfiguration du protocole de validation est basée sur l'équation suivante. Cette équation compare le coût cumulé de la validation et de l'abandon de 100 transactions en fonction du taux de validation :

$$\begin{cases} x + y = 100 \\ x \times C_{2PC-PC} + y \times A_{2PC-PC} < x \times C_{2PC-PA} + y \times A_{2PC-PA} \end{cases}$$

Dans cette équation, x (*resp.* y) représente le nombre de transactions validées (resp. abandonnées) et C_{2PC-PX} (*resp.* A_{2PC-PX}) représente le coût de la validation (resp. abandon) du protocole 2PC-PX. Cette équation détermine le taux représentant la limite pour laquelle le protocole 2PC-PC est plus adapté que le protocole 2PC-PA. La solution de cette équation est :

$$\begin{cases} y = 100 - x \\ x > \frac{100 \times (A_{2PC-PA} - A_{2PC-PC})}{(C_{2PC-PC} - A_{2PC-PC} - C_{2PC-PA} + A_{2PC-PA})} \end{cases}$$

La figure 11.9 applique cette solution sur les mesures réalisées dans la section 11.3.1 afin de déterminer la valeur du taux limite. Ce taux varie en fonction du nombre de participants impliqués dans la transaction. Par exemple, pour une transaction dans laquelle sont impliqués 20 participants, le protocole 2PC-PC est plus adapté que le protocole 2PC-PA pour un taux de validation supérieur à 54%. Ce taux limite est utilisé par CATE comme condition de la reconfiguration du protocole de validation.

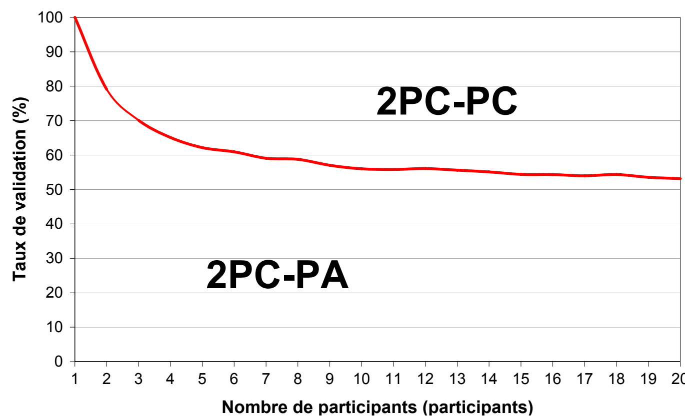


FIG. 11.9: Taux limite de reconfiguration du protocole de validation.

La figure 11.10 présente l'architecture du composant Context Awareness représentant la politique d'adaptation du service de transactions. Cette politique est composée de deux parties. Le

composant Commit Rate calcule le taux limite courant en fonction du nombre moyen de participants par transaction. Si ce taux varie, le composant Commit Rate reconfigure le composant Commit Protocol pour préciser la nouvelle valeur du taux limite. Le composant Commit Protocol observe quant à lui le taux de validation des transactions exécutées. Le calcul de ce taux de validation est basé sur une fenêtre de visibilité de taille variable. Ce calcul évite des reconfigurations abusives du protocole de validation lorsque que le taux de validation/abandon fluctue de façon trop importante.

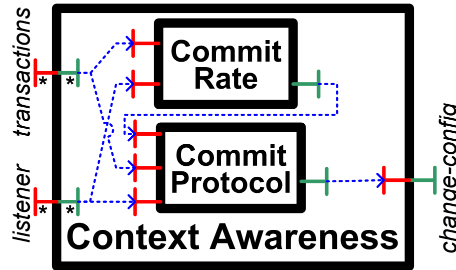
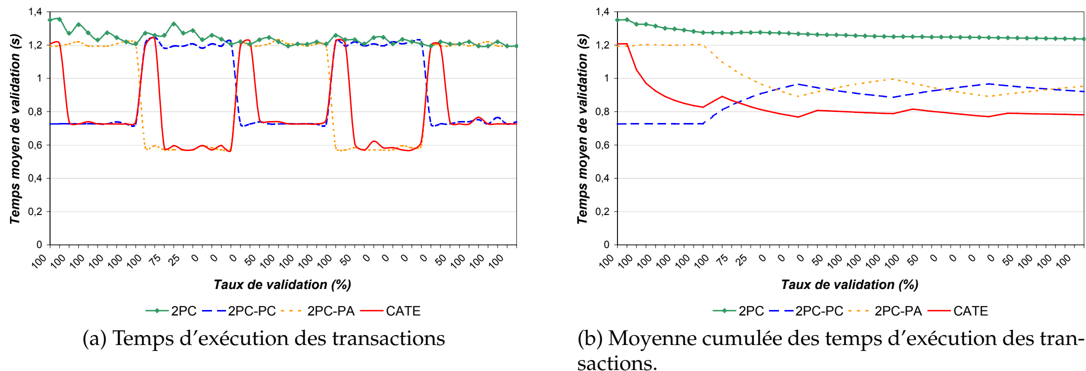


FIG. 11.10: Architecture de la politique d'adaptation.

11.4.4 Évaluation empirique des performances de CATE

CATE n'utilise jamais le protocole 2PC lors de la reconfiguration car celui-ci n'offre pas de meilleures performances que les protocoles 2PC-PC et 2PC-PA quel que soit le contexte d'exécution. Le scénario des figures 11.11a et 11.11b évalue le temps moyen de validation de 50 transactions exécutées séquentiellement dans un contexte d'exécution variant de façon constante (10 transactions valident puis 10 transactions abandonnent puis 10 transactions valident, etc.). Des services de transactions configurés avec un protocole de validation particulier sont également évalués dans ces conditions et comparés à CATE. La figure 11.11b présente l'historique du temps d'exécution moyen des transactions depuis le démarrage du service de transactions.



(a) Temps d'exécution des transactions

(b) Moyenne cumulée des temps d'exécution des transactions.

FIG. 11.11: Évaluations des performances de CATE.

Les mesures présentées dans la figure 11.11a montrent la variation du temps d'exécution des transactions en fonction du taux de validation/abandon des transactions. Dans ce contexte, CATE obtient de meilleures performances que les services de transactions configurés avec un seul protocole de validation grâce à ses capacités d'auto-adaptation. En effet, les performances des services de transactions utilisant les protocoles 2PC, 2PC-PC et 2PC-PA sont impactées par les variations du taux de validation/abandon. Dans CATE, lorsque le taux de validation est élevé, le protocole

de validation utilisé est 2PC-PC alors que CATE utilise le protocole 2PC-PA lorsque ce taux est inférieur au taux limite. Par conséquent, CATE bénéficie à tout moment du meilleur des deux protocoles. Les mesures présentées dans la figure 11.11b montrent que sur le long terme, CATE fournit des performances meilleures que les autres services. Les reconfigurations de CATE ne sont effectuées que lorsque le taux limite est franchi. De plus le composant Tx Factory intègre une fonctionnalité de cache (présentée dans le chapitre 7) afin de réduire le coût lié au changement de configurations et la création des instances de transactions.

11.4.5 Discussions

Dans cette section nous discutons différents aspects relatifs à l'utilisation de CATE.

Reconfiguration des transactions actives. Changer le protocole des transactions en cours d'exécution peut compromettre le processus de reprise sur faute de la transaction en cas d'échec. C'est pour cette raison que CATE n'autorise pas la reconfiguration des transactions en cours d'exécution. Ainsi, des transactions peuvent s'exécuter avec des protocoles de validation différents mais lorsqu'une transaction a débuté, son protocole de validation ne peut plus être changé.

Utilisation de CATE. Afin d'utiliser CATE dans un serveur d'applications par exemple, il est nécessaire de garantir certaines hypothèses. 1) La partie *participant* est gérée par les gestionnaires de ressources dont l'implantation est indépendente de celle du service de transactions. L'architecture présentée dans la figure 11.8 est une solution possible. 2) Les protocoles gérés par CATE doivent également être supportés par les gestionnaires de ressources. 3) Les gestionnaires de ressources doivent être capables de changer dynamiquement le protocole de validation.

Extension de CATE. CATE peut supporter d'autres protocoles de validation que ceux présentés dans ce chapitre. Le choix des protocoles de type 2PC nous a permis de présenter un exemple de réutilisation des composants entre les différentes versions des protocoles. Néanmoins, la réutilisation des composants n'est pas un prérequis de CATE et d'autres protocoles peuvent être pris en compte. Par exemple, les protocoles 1PC et 3PC peuvent être considérés par CATE dans la mesure où l'auto-adaptation du protocole se base sur les effets observables des transactions. Cependant, l'utilisation de ces protocoles requiert que l'application prenne en charge les principes associés à ces protocoles.

Issues prévisibles. Certains protocoles tirent parti des issues prévisibles des transactions [AHCL97, AS02, YWP04]. En utilisant des mécanismes de *piggybacking* ou de *callback*, ces protocoles peuvent déterminer si les transactions sont marquées en lecture seule ou pour l'abandon avant de débuter le protocole de validation. Par conséquent, la connaissance de certaines issues de la transaction permet d'optimiser le temps d'exécution de la transaction. Les protocoles définis dans ces approches restent complémentaires à notre proposition car CATE s'attache à optimiser le temps d'exécution des transactions aux issues imprévisibles.

Préservation de la sémantique globale du système. Dans le domaine de la reconfiguration logicielle, il est nécessaire de préserver la sémantique du système. Dans notre cas, ce sont les propriétés transactionnelles qui doivent être préservées. Ainsi, si un protocole de validation atomique est remplacé par un autre protocole de validation ne respectant pas cette même propriété (par exemple, l'atomicité sémantique [GMS83]), l'intégrité de la transaction peut être compromise. C'est pour cette raison que ce chapitre se concentre uniquement sur les protocoles du type 2PC sémantiquement équivalents.

11.5 Travaux connexes

Dans [DL02], les auteurs proposent d'adapter dynamiquement des préoccupations fonctionnelles (relatives à l'application) et non-fonctionnelles par un mécanisme de tissage à l'exécution.

Les auteurs s'intéressent à rendre le processus de tissage adaptable au contexte d'exécution. Leur objectif est de pouvoir choisir dynamiquement le code non-fonctionnel approprié. Leur proposition est de définir des politiques d'adaptation sensibles au contexte (en utilisant un mécanisme de règles ECA) et de tisser automatiquement ces politiques.

ReflectTS propose un mécanisme d'adaptation à l'exécution pour l'assemblage de services non-fonctionnels en utilisant des dépôts de services [AK05]. Ces dépôts hébergent les services non-fonctionnels sous forme de composants et leurs descriptions sous forme de méta-données. Cette approche requiert que l'application elle-même soit développée sous forme de composants. Notre approche n'a aucun prérequis vis-à-vis de l'application et nous préférons adapter le service non-fonctionnel que l'application qui utilise le service non-fonctionnel.

Comparés à notre approche, [DL02] et ReflectTS considèrent le service non-fonctionnel comme granularité d'adaptation. Notre approche propose d'intégrer des propriétés d'auto-adaptation dans le service non-fonctionnel en utilisant les composants qui le composent comme granularité d'adaptation. Notre approche est validée sur un exemple démontrant l'intérêt et le bénéfice de l'auto-adaptation pour un service de transactions.

Le protocole de validation introduit dans [YWP04] propose de supporter la notion de *web services* auto-adaptables qui peuvent supporter des participants utilisant aussi bien les protocoles 2PC-PA et 2PC-PC. Ce protocole permet donc de combiner dynamiquement dans une même transaction des participants utilisant des protocoles de validation différents. Comparé à notre proposition, ce protocole ne s'intéresse pas à l'évolution du protocole en fonction du contexte d'exécution mais il fournit un moyen de combiner des protocoles hétérogènes. Dans notre approche, nous considérons les protocoles 2PC, 2PC-PC et 2PC-PA comme des exemples de protocoles et nous pouvons en intégrer de nouveaux.

En général, les travaux présentés dans [MLO86, CSAH98, LAA98] sont des travaux basés sur la simulation. Les évaluations de performances se concentrent sur la sémantique des transactions (par exemple, transaction en lecture seule, transaction de mise à jour) et sur la présence d'erreurs. Dans ce chapitre, nous nous intéressons au coût d'exécution des protocoles de validation en fonction du taux de validation/abandon et au coût de l'utilisation d'un mécanisme d'auto-adaptation pour choisir le protocole de validation le plus adapté au contexte d'exécution. Notre implantation de ce mécanisme et son évaluation montrent que le coût de la reconfiguration peut être négligé face au gain de performance induit par l'utilisation du protocole le plus adapté.

11.6 Conclusion

Si l'auto-adaptation représente un point d'intérêt de la communauté du développement logiciel, peu de travaux se sont penchés sur l'auto-adaptation au niveau des services intergiciels [AK05, Hér05]. Ce chapitre s'intéresse à l'auto-adaptabilité des services de transactions et plus particulièrement de leur protocole de validation.

D'une part, nous proposons une implantation sous forme de composants des protocoles de validation 2PC standards. Chaque protocole utilise les mêmes composants mais ceux-ci sont connectés différemment entre eux. D'autre part, nous proposons un service de transactions sensible au contexte appelé CATE. CATE choisit le protocole de validation le plus approprié au contexte d'exécution. L'évaluation de performances montre que l'auto-adaptation du protocole de validation donne de meilleurs performances que l'utilisation d'un service ne supportant qu'un seul protocole de validation.

Les perspectives liées à ce travail adressent la prise en compte de nouveaux protocoles de validation de type 2PC mais aussi 1PC et 3PC. L'évaluation empirique des performances de ces protocoles semble intéressante afin d'étendre les politiques d'adaptation de CATE.

Les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Patricia Serrano-Alvarado et ils ont donné lieu à deux publications internationales [SARM05, RSAM06b].

If it works, it's production. If it doesn't, it's a test.

Philington's First Law

Chapitre 12

Conclusion

Sommaire

12.1 Principaux apports **209**
12.1.1 Démarche extrêmement fine de construction de canevas intergiciels . . . 210
12.1.2 Canevas de construction de services de transactions hautement adaptables 211
12.2 Publications **211**
12.2.1 Revues internationales 211
12.2.2 Conférences internationales 212
12.2.3 Ateliers internationaux 212
12.2.4 Conférence nationale 212
12.2.5 Atelier national 212
12.3 Perspectives **213**
12.3.1 Axes de développement 213
12.3.2 Axes de recherche 213

DANS CE CHAPITRE, NOUS DRESSONS LE BILAN DES TRAVAUX *réalisés dans le cadre de cette thèse. Nous y présentons les principaux apports de notre proposition dans le cadre de la construction des canevas intergiciels hautement adaptables et plus précisément des services de transactions. Ces travaux ont pu être validés par des publications dans des conférences ou des ateliers d'audience internationale. Enfin, nous présentons les perspectives relatives à l'amélioration des différentes contributions proposées dans cette thèse.*

12.1 Principaux apports

L'expansion et la diversification des standards transactionnels nécessite de fournir des abstractions fonctionnelles et architecturales de la fonction transactionnelle afin de réduire les coûts et les temps de développement des implantations de ces standards. Ces services de transactions doivent également prendre en compte de nouvelles préoccupations pour s'adapter à l'hétérogénéité et la dynamicité des contextes d'exécution dans lesquels ils peuvent être déployés. Pour faire face à ces besoins constants d'adaptation, nous proposons une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables, et nous appliquons cette démarche à la construction des services de transactions.

12.1.1 Démarche extrêmement fine de construction de canevas intergiciels

Dans cette thèse, nous avons complété les principes de construction des exogiciels défini par [Qué05] pour faciliter la construction d'architectures basées sur des composants de granularité très fine. Nous avons non seulement utilisé le modèle de composants FRACTAL et le langage d'architecture FRACTAL ADL, mais nous avons aussi proposé deux extensions généralistes pour faciliter le développement d'applications avec ce modèle :

- *Le modèle de programmation* FRACLET facilite la programmation des composants FRACTAL en réduisant l'impact des préoccupations techniques dans le code des composants primitifs. Pour ce faire, nous définissons un ensemble d'annotations réifiant les principales notions définies dans un modèle de composants. Ces annotations sont ensuite consommées par des générateurs pour produire automatiquement non seulement le code technique adapté au composant, mais aussi les descriptions d'architecture qui lui sont associées. Ainsi, le développement d'une application basée sur le modèle de composants FRACTAL gagne en rapidité et en fiabilité grâce à FRACLET. De plus, nous avons montré que notre modèle de programmation pouvait être également utilisé avec d'autres modèles de composants comme le modèle OPENCOM.
- *Le langage de description et de vérification des motifs d'architecture*, basé sur FRACTAL ADL, permet de décrire et de vérifier l'application de motifs d'architecture. Nous proposons d'identifier et d'isoler les motifs d'assemblage récurrents dans une architecture à l'aide du langage FPATH. En effet, le langage FPATH fournit une syntaxe concise permettant de capturer n'importe quel élément caractéristique d'une architecture à base de composants. Ainsi, nous intégrons ce langage dans FRACTAL ADL afin de générer automatiquement des motifs d'assemblage en fonction des composants présents dans l'architecture à l'aide de l'opérateur `foreach`. Le second opérateur `assert` utilise également le langage FPATH pour vérifier des invariants architecturaux lors du déploiement d'une application. Ces mêmes invariants architecturaux sont également garantis lors de reconfigurations *ad-hoc* de l'architecture par l'extension du contrôle des composants FRACTAL.

Ces deux contributions généralistes au modèle de composants FRACTAL constituent un support fiable pour la construction de canevas intergiciels hautement adaptables. Si la définition de ces canevas est très liée à leur domaine d'application, nous pouvons néanmoins énoncer trois principes de conception facilitant la construction de canevas intergiciels hautement adaptables :

- *La programmation des composants doit adopter une granularité atomique*. Cette granularité très fine accroît les capacités d'adaptation du canevas intergiciel et facilite la réutilisation des composants entre les personnalités.
- *La structuration de l'architecture doit s'appuyer sur des motifs de conception*. Les motifs de conception identifient des problèmes communément rencontrés et fournissent une solution élégante pour les résoudre. Notamment, les patrons de conception facilitent l'évolution des architectures en offrant des solutions extensibles.
- *La configuration du canevas doit reposer sur des modèles de haut niveau*. Les modèles de haut niveau permettent de capitaliser les fonctionnalités complexes d'un système indépendemment d'une mise en œuvre particulière. Ils peuvent également fournir une sémantique plus riche que les descripteurs d'architecture traditionnels afin de faciliter certaines vérifications plus formelles.

Nous avons ensuite appliqué ces principes de conception dans le cadre de la définition du canevas intergiciel GOTM. Ce canevas est destiné à la construction de services de transactions hautement adaptables.

12.1.2 Canevas de construction de services de transactions hautement adaptables

La définition du canevas GoTM et la construction de différentes personnalités nous a permis de valider nos principes de conception. En particulier, nous avons pu démontrer qu'il est possible de se détacher des abstractions architecturales imposées par les standards transactionnels en appliquant les motifs de conception ADAPTER et COMMAND. De même, l'application du motif de conception STRATEGY a permis de supporter différents protocoles de validation des transactions tandis que le motif STATE a fourni une représentation architecturale de l'automate d'états d'une transaction. D'autre part, le service de transaction a été structuré à l'aide de trois motifs de conception. Ainsi, le motif de conception FACTORY s'appuie sur le motif PROTOTYPE pour créer les instances de transactions. Ces instances sont alors gérées par le motif de conception SINGLETON.

Dans un second temps, nous avons défini différents modèles de haut niveau afin de faciliter la configuration des composants réalisant ces motifs de conception. Par exemple, le motif de conception STRATEGY est configurable par un diagramme UML de séquences alors que le diagramme UML d'états nous permet de configurer le motif de conception STATE. Nous utilisons également des règles ECA pour générer et assembler le composant réalisant le motif de conception COMMAND. Enfin, nous avons défini un méta-modèle dédié à l'assemblage d'un service de transactions. Ce méta-modèle s'inspire des motifs de conception que nous avons identifiés et permet de coordonner le processus de construction du service de transactions en transformant les configurations décrites dans les différents modèles de haut niveau vers des assemblages de composants concrets.

Enfin, nous avons procédé à la réalisation de personnalités de services de transactions originales afin de valider les propriétés d'adaptabilité du canevas GoTM. Nous avons d'abord construit la personnalité JOTDF de service de transactions intégrant des politiques de démarcation transactionnelle afin de prendre en charge automatiquement cette fonction requise par les plates-formes d'exécution à base de composants. Puis, nous avons construit la personnalité ATS de service de transactions supportant plusieurs standards transactionnels simultanément. Ce type de construction permet d'intégrer de façon transparente le support transactionnel d'applications patrimoniales développées avec des technologies différentes. Finalement, nous avons construit la personnalité CATE de service de transactions intégrant plusieurs protocoles de validation. Cette expérience nous a permis de sélectionner dynamiquement le protocole de validation le plus performant en observant le contexte d'exécution et en reconfigurant l'usine de transactions automatiquement.

Toutes ces expériences ont non seulement démontré l'intérêt de notre proposition mais elles ont également prouvé que l'application du paradigme de composant à granularité extrêmement fine dans la réalisation des fonctions intergicielles n'introduisait pas nécessairement de surcoût à l'exécution. Au contraire, nous observons dans la plupart de nos expériences un léger gain par rapport aux implantations de services de transactions performants (par exemple, JOTM).

12.2 Publications

Cette section recense les différents articles que nous avons publiés dans le cadre de cette thèse.

12.2.1 Revues internationales

1. Romain Rouvoy et Philippe Merle, *Using Microcomponents and Design Patterns to Build Evolutionary Transaction Services*, International ERCIM Workshop on Software Evolution, ENCTS (2006), à paraître. Travaux présentés dans les chapitres 7 et 8.

2. Christophe Demarey, Gael Harbonnier, Romain Rouvoy et Philippe Merle, *Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms*, Studia Informatica Universalis Regular Issue **4** (2005), no. 1, 7–24.

12.2.2 Conférences internationales

1. Romain Rouvoy, Patricia Serrano-Alvarado et Philippe Merle, *Towards Context-Aware Transaction Services*, 6th International IFIP Conference on Distributed Applications and Interoperable Systems (Bologna, Italia), LNCS, vol. 4025, Springer, Juin 2006, pp. 272–288. Travaux présentés dans les chapitres 7, 8 et 11.
2. Romain Rouvoy, Patricia Serrano-Alvarado et Philippe Merle, *A Component-based Approach to Compose Transaction Standards*, 5th International ETAPS Symposium on Software Composition (Vienna, Austria), LNCS, vol. 4089, Springer, Mars 2006, pp. 114–130. Travaux présentés dans les chapitres 7, 8 et 10.
3. Marek Procházka, Romain Rouvoy et Thierry Coupaye, *On Enhancing Component-Based Middleware with Transactions*, 5th International Symposium on Distributed Object and Applications (Catania), LNCS, vol. 2889, Springer–Verlag, Novembre 2003, Poster session, pp. 1–2. Travaux présentés dans le chapitre 9.
4. Romain Rouvoy et Philippe Merle, *Abstraction of Transaction Demarcation in Component-Oriented Platforms*, 4th ACM/IFIP/USENIX International Middleware Conference (Rio de Janeiro, Brasil), LNCS, vol. 2972, Springer, Juin 2003, pp. 305–323. Travaux présentés dans le chapitre 9.

12.2.3 Ateliers internationaux

1. Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak et Philippe Merle, *Using Attribute-Oriented Programming to Leverage Fractal-Based Developments*, 5th International ECOOP Workshop on Fractal Component Model (Nantes, France), Juillet 2006. Travaux présentés dans le chapitre 5.
2. Romain Rouvoy et Philippe Merle, *Leveraging Component-Oriented Programming with Attribute-Oriented Programming*, 11th International ECOOP Workshop on Component-Oriented Programming (Nantes, France), Juillet 2006. Travaux présentés dans le chapitre 5.
3. Patricia Serrano-Alvarado, Romain Rouvoy et Philippe Merle, *Self-Adaptive Component-Based Transaction Commit Management*, 4th Middleware Workshop on Adaptive and Reflective Middleware (Grenoble, France), AICPS, vol. 116, ACM, Novembre 2005, pp. 1–6. Travaux présentés dans les chapitres 7 et 11.
4. Romain Rouvoy et Philippe Merle, *Towards a Model Driven Approach to Build Component-Based Adaptable Middleware*, 3rd Middleware Workshop on Reflective and Adaptive Middleware (Toronto, Ontario, Canada), AICPS, vol. 80, ACM, Octobre 2004, pp. 195–200. Travaux présentés dans le chapitre 8.

12.2.4 Conférence nationale

1. Romain Rouvoy et Philippe Merle, *GoTM : vers un canevas transactionnel à base de composants*, 10ème Conférence sur les Langages, Modèles & Objets (Lille, France), vol. 10, L'Objet, no. 1-3/2004, Hermès Science, Mars 2004, pp. 131–146. Travaux présentés dans le chapitre 7.

12.2.5 Atelier national

1. Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak et Philippe Merle, *Apports de la Programmation par Attributs au Modèle de Composants Fractal*, 5èmes Journées Composants (Perpignan, France), Octobre 2006, à paraître. Travaux présentés dans le chapitre 5.

12.3 Perspectives

Dans cette section, nous introduisons les perspectives qui s'ouvrent à l'issue de cette thèse. Nous discutons dans un premier temps la nécessité de poursuivre le développement des canevas que nous avons présentés dans ce manuscrit puis nous détaillons différentes possibilités quant à l'extension de notre proposition.

12.3.1 Axes de développement

Langage de description et de vérification des architectures. Le langage de description et de vérification d'architecture présenté dans le chapitre 6 est à l'heure actuelle partiellement réalisé. Nous disposons d'une implantation minimaliste de l'interprêteur FPATH sous forme de composants FRACTAL nous permettant de naviguer dans une architecture disponible à l'exécution. Nous disposons également d'un opérateur FRACTAL ADL opérant des transformations configurables de l'AST FRACTAL. Par conséquent, il nous reste désormais à réaliser l'intégration de notre implantation de FPATH dans une version spécialisée de l'opérateur de transformation afin de supporter la définition des motifs d'architecture dans FRACTAL ADL.

Transformations des modèles de haut niveau. De la même façon, les outils de transformation des modèles de haut niveau présentés dans le chapitre 8 ne sont pas encore totalement implantés. Si nous avons identifié exhaustivement les règles de transformation permettant de produire un descripteur d'architecture FRACTAL ADL à partir de diagrammes UML ou de règles ECA, il nous reste encore à implanter l'opération de transformation en nous appuyant sur un moteur de transformation existant afin d'automatiser le processus.

Évaluation et comparaison des performances. Nous avons pu observer des propriétés de performances pertinentes lorsque nous avons comparé les services de transactions construits avec GOTM et le service de transactions JOTM [Mes03]. Nous souhaitons approfondir ces évaluations de performances en utilisant un scénario de *benchmark* reconnu tel que TPC-C[1] ou RUBiS[2] afin de nous comparer au service de transactions ARJUNATS (présenté dans le chapitre 2) qui a récemment été intégré dans le serveur J2EE JBoss.

12.3.2 Axes de recherche

Modèle de programmation

Le modèle de programmation FRACLET présenté dans le chapitre 5 peut être amélioré via l'intégration de trois travaux.

Modularisation et extension de FRACLET. Le modèle de programmation FRACLET supporte à l'heure actuelle la programmation de composants exécutables sur des modèles de composants tels que Fractal ou OpenCOM. Nous souhaiterions étendre les capacités de FRACLET non seulement pour prendre en compte de nouveaux modèles de composants mais aussi pour faciliter le support des propriétés non-fonctionnelles. Parmi les extensions possibles, nous pouvons citer la définition d'annotations pour la démarcation des transactions (présenté dans le chapitre 9) ou le typage des assemblages de composants [BLQ+05]. Le défi relatif à ces extensions consiste à maximiser la modularité des extensions afin de minimiser leurs dépendances.

[1] Le *benchmark* TPC-C : http://www.tpc.org/tpcc/
[2] Le *benchmark* RUBiS : http://rubis.objectweb.org/

Vérification du code des composants. En marquant le code source des composants avec les annotations définies par FRACLET, il devient possible de vérifier que le code source respecte les règles de programmation imposées par le modèle de composants. Pour l'instant, nous sommes en mesure de vérifier que la déclaration de la référence d'une interface requise correspond bien à une interface et que les attributs de composants sont de type primitif. Nous souhaiterions poursuivre ces vérifications afin de détecter les chemins de communication cachés dans une architecture ou les reconfigurations non-autorisées des caractéristiques du composant.

Programmation des composants par *mixins*. L'application de la programmation par attributs au modèle de composants a permis de faciliter le développement de composants de très fine granularité. Nous souhaiterions poursuivre cette démarche en intégrant le principe de programmation par *mixins* [BCL+06] dans le développement des composants. En effet, les *mixins* proposent une granularité extrême permettant de diviser le code source d'un composant en différents modules pour les sélectionner et les fusionner *a posteriori*. Ce type de construction pourrait bénéficier aux approches structurant la membrane des composants métiers sous la forme d'assemblages de composants techniques (les contrôleurs) telles que AOKELL [SPDC06]. L'intégration de nouveaux contrôleurs dans ces membranes pourrait utiliser les mixins pour introduire la déclaration des interfaces clientes et du code d'invocation dans le code source des contrôleurs existants.

Langage de description d'architectures

En nous appuyant, sur le langage de description et de vérification des motifs décrit dans le chapitre 6, nous pouvons imaginer deux travaux potentiellement intéressants à mener.

Langage de description d'architecture générique. Nous avons pu observer que le langage de description d'architectures FRACTAL ADL est particulièrement extensible grâce notamment à sa syntaxe basée sur le langage XML et à son usine à base de composants. Dès lors, nous souhaiterions étudier la possibilité d'utiliser FRACTAL ADL comme un langage de description d'architectures pivot. En effet, notre modèle de programmation FRACLET est suffisamment abstrait pour supporter différents modèles de composants et celui-ci est capable de générer automatiquement les descriptions FRACTAL ADL associées aux composants primitifs. Par conséquent, nous souhaiterions nous appuyer sur les capacités de génération de FRACLET et l'ouverture de l'usine FRACTAL ADL pour supporter le déploiement d'autres modèles de composants que le modèle FRACTAL. Ce type de support nécessite la définition d'un nouveau composant *backend* (le mécanisme d'interprétation des constructions FRACTAL ADL) pour chaque modèle de composants cible. De plus, ce type d'approche permettrait de faire bénéficier à d'autres modèles de composants des extensions développées pour la syntaxe FRACTAL ADL (par exemple, motifs d'architecture, architecture patrimoniale [Qué05], composants d'aspects [PSCD06]).

Optimisation de l'empreinte mémoire. Les expériences réalisées avec les personnalités de services de transactions construites à partir du canevas GOTM ont montré qu'une approche à granularité très fine n'introduisait pas de surcoût à l'exécution. Cependant, la multiplication des composants dans une architecture peut impacter son empreinte mémoire par la multiplication du code technique et des méta-informations relatives à l'architecture. La prise en compte des contextes d'exécution contraints peut cependant nécessiter de réduire la taille de cette empreinte mémoire. L'implantation JULIA [BCL+06] permet à l'heure actuelle de fusionner le code des contrôleurs et des intercepteurs avec le contenu des composants mais ce type d'optimisation peut ne pas suffir pour une exécution en environnement contraint. Sans remettre en cause les composants de fine granularité définis dans le canevas GOTM, nous souhaiterions mettre en place un mécanisme de fusion des composants lors de l'assemblage de l'architecture. L'objectif de ce mécanisme est de pouvoir transformer un composant composite contenant plusieurs composants en un composant primitif afin de réduire son empreinte mémoire. Ce type de fusion

nécessite de manipuler le code source d'un ensemble de composants (ou leur code compilé) pour en inférer la définition d'un composant monolithique équivalent.

Canevas de composants GOTM

Le canevas de composants GOTM présenté dans le chapitre 7 fournit un ensemble de composants permettant de construire des services de transactions basés sur les modèles de transactions plats. Deux travaux d'extension de ce canevas permettraient de construire des services de transactions plus évolués.

Support des modèles de transactions étendus. Les modèles de transactions étendus n'ont volontairement pas été abordés dans cette thèse afin de nous concentrer sur la structuration du cœur d'un service de transactions. De nombreux travaux présentés dans le chapitre 2 offrent des abstractions architecturales et fonctionnelles pertinentes pour la prise en compte de plusieurs modèles de transactions étendus. Dans le cadre d'un rapprochement du canevas GOTM avec le service de transactions ATPMOS (développé pour KALA), nous souhaitons intégrer de nouveaux motifs de conception et composants pour supporter d'autres modèles de transactions que le modèle plat. Le support de ces modèles de transactions étendus devrait nous permettre de supporter d'autres standards transactionnels tels que les standards *Web Services Business Activities* [CCF+05c] ou *Additional Structural Mechanisms for OTS* [OMG05a].

Passage à l'échelle des services de transactions GOTM. Dans un contexte où les environnements d'exécution se diversifient et atteignent des dimensions supérieures aux réseaux d'entreprises, nous souhaitons étudier la conception de services de transactions GOTM adaptés aux dimensions et aux particularités des réseaux de grilles de calcul ou des réseaux ubiquitaires. Ces particularités nous permettraient notamment d'étudier la modélisation et la réalisation des protocoles de validation en deux phases hiérarchiques ou distribués dans le cadre des grilles de calcul. Dans le contexte des réseaux ubiquitaires, nous pourrions utiliser GOTM pour mettre en œuvre et expérimenter différents protocoles de validation pour l'informatique mobile tels que les protocoles TCOT (*Timeout-based Mobile Transaction Commitment*), M2PC (*Mobile 2PC*), CO2PC (*Combination of an Optimistic approach and 2PC*) ou UCM (*Unilateral Commit for Mobile Environment*) [SARA04].

Support des modèles de concurrence. De façon similaire, nous n'avons pas adressé les protocoles de concurrence dans notre étude. Cependant, les résultats que nous avons obtenus en terme de modularité et de performance nous encouragent à appliquer les principes de conception à très fine granularité sur d'autres fonctions intergicielles. La gestion de la concurrence étant liée de prêt aux préoccupations transactionnelles, l'intégration de nouveaux composants dédiés à cette préoccupation dans le canevas GOTM permettrait de compléter notre solution.

Modèles de configuration de GOTM

Les modèles de haut niveau présentés dans le chapitre 8 facilitent la construction des personnalités de services de transactions avec GOTM. Nous souhaiterions poursuivre ce travail d'abstraction et de vérification de la fonction transactionnelle au travers de deux travaux.

Typage des transactions. Nous avons présenté différentes configurations et reconfigurations des architectures des services de transactions dans cette thèse. Cependant, GOTM ne fournit aucun système de typage permettant de vérifier la validité et la cohérence de ces assemblages. Néanmoins, GOTM est capable de réifier architecturalement l'automate d'état d'une transaction sous la forme d'un assemblage de composants (présenté dans le chapitre 7). Dès lors, nous souhaiterions étudier si la théorie des systèmes de typage comportementaux [NNS01] peuvent

nous permettre de formaliser la définition d'une transaction en nous appuyant sur son automate d'états. Ainsi, la reconfiguration d'une fonctionnalité modifiant l'état d'une transaction (par exemple, le protocole de validation) avec une nouvelle stratégie pourrait vérifier que leurs automates d'états respectifs sont compatibles avant d'effectuer la reconfiguration concrète de la transaction.

Définition d'un langage dédié pour les services de transactions. L'utilisation des modèles de haut niveau nous a permis de construire des personnalités de services de transactions dans la limite des capacités offertes par les modèles que nous utilisons (UML, ECA). Nous souhaiterions étudier la définition d'un langage dédié à la construction des services de transactions en nous inspirant du canevas ACTA [CR90]. Notre objectif est de fournir une syntaxe précise et concise pour décrire les caractéristiques d'un service de transactions et construire automatiquement ce services avec les composants mis à disposition par le canevas GOTM. Ce modèle dédié permettrait de réduire le fossé existant entre la formalisation des modèles de transactions et leur implantation de façon analogue aux travaux qui ont pu être réalisés par KALA sur la démarcation des transactions [Fab05].

Généralisation de notre démarche à d'autres services intergiciels.

Nous pensons que la démarche à granularité extrêmement fine que nous avons présentée dans cette thèse peut être généralisée à la construction d'autres services intergiciels hautement adaptables. Nous avons déjà commencé à appliquer les principes généralistes et les outils de notre démarche à d'autres canevas intergiciels développés avec le modèle de composants FRACTAL.

Canevas de composition des ressources système. Le canevas intergiciel COSMOS est dédié à la réification et à la composition de ressources système [Con06]. Dans ce canevas, les ressources système sont réifiés sous la forme de composants qui décrivent l'état courant du périphérique en utilisant des messages DREAM. COSMOS utilise intensivement notre modèle de programmation FRACLET et notre langage de description et de vérification des motifs d'architectures respectivement pour réifier les ressources système et les composer. COSMOS identifie également un certain nombre de motifs de conception qui simplifient la description de la composition des ressources systèmes pour en inférer des situations d'adaptation. L'application de notre démarche dans le cadre de COSMOS a permis de simplifier le processus de configuration du canevas COSMOS en favorisant l'identification des attributs des composants et en fournissant un mécanisme de configuration de ces derniers.

Canevas de déploiement des intergiciels. Le canevas intergiciel FRACTAL DEPLOYMENT FRAMEWORK (FDF) est dédié au déploiement de systèmes distribués [FM06]. Dans ce canevas, le système cible (la machine, l'utilisateur, le protocole d'accès, etc.) comme l'application (les composants applicatifs, la plate-forme intergicielle, le substrat d'exécution, etc.) sont décrits en utilisant le langage FRACTAL ADL. FDF utilise intensivement notre modèle de programmation FRACLET et notre langage de description et de vérification des motifs d'architecture respectivement pour réifier les caractéristiques d'un système cible et modéliser un système cible particulier. L'application de notre démarche permet de modéliser très finement les caractéritiques d'un système cible et d'y déployer n'importe quelle infrastructure logicielle.

Bibliographie

[Abd06] Takoua Abdellatif Berrayana, *Apport des architectures à composants pour l'administration des intergiciels. Étude de cas : JonasALaCarte, un serveur d'applications J2EE administrable*, Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, Septembre 2006.

[ACBD+04] Mourad Alia, Sebastien Chassande-Barrioz, Pascal Dechamboux, Catherine Hamon, et Alexandre Lefebvre, *A Middleware Framework for the Persistence and Querying of Java Objects*, 18th European Conference on Object-Oriented Programming (Oslo, Norway), LNCS, vol. 3086, Springer, Juin 2004, pp. 291–315.

[ACN02a] Jonathan Aldrich, Craig Chambers, et David Notkin, *Architectural Reasoning in ArchJava*, 16th European Conference on Object-Oriented Programming (Málaga, Spain), LNCS, vol. 2374, Springer, Juin 2002, pp. 334–367.

[ACN02b] ———, *ArchJava : Connecting Software Architecture to Implementation*, 24th Int. Conference on Software Engineering (Orlando, Florida, USA), ACM, Mai 2002, pp. 187–197.

[ACV97] Eman Anwar, Sharma Chakravarthy, et Marisa S. Viveros, *An Extensible Approach to Realizing Advanced Transaction Models*, Advanced Transaction Models and Architectures, Kluwer, 1997, pp. 259–276.

[AHCL97] Yousef J. Al-Houmaily, Panos K. Chrysanthis, et Steven P. Levitan, *Enhancing the Performance of Presumed Commit Protocol*, 12th ACM Symposium on Applied Computing (San Jose, California, USA), ACM, Février 1997, pp. 131–133.

[AK05] Anna-Brith Arntsen et Randi Karlsen, *ReflecTS : a Flexible Transaction Service Framework*, 4th Int. Middleware Workshop on Adaptive and Reflective Middleware (Grenoble, France), ACM Int. Conference Proceeding Series, vol. 116, ACM, Novembre 2005, pp. 1–6.

[All97] Robert Allen, *A Formal Approach to Software Architecture*, PhD dissertation, Carnegie Mellon, School of Computer Science, Janvier 1997, Issued as CMU Technical Report CMU-CS-97-144.

[Arj03a] Arjuna, *Introducing WS-Coordination*, Technical report, Arjuna Technologies Ltd., Newcastle, UK, Avril 2003.

[Arj03b] ———, *Introducing WS-Transaction*, Technical report, Arjuna Technologies Ltd., Newcastle, UK, Mai 2003.

[AS02] Gopi K. Attaluri et Kenneth Salem, *The Presumed-Either Two-Phase Commit Protocol*, IEEE Transactions on Knowledge and Data Engineering **14** (2002), no. 5, 1190–1196.

[Bar98] Roger S. Barga, *A Reflective Framework for Implementing Extended Transactions*, PhD dissertation, Oregon Graduate Institute, Portland, Oregon, USA, 1998.

[Bar05] Olivier Barais, *Construire et Maîtriser l'Évolution d'une Architecture Logicielle à base de Composants*, Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, Novembre 2005.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, et Romain Quilici, *Grid Computing : Software Environments and Tools*, ch. Programming, Deploying, Composing, for the Grid, Springer, Janvier 2006.

[BCA+01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fábio M. Costa, Hector A. Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui S. Moreira, Nikos Parlavantzas, et Katia B. Saikoski, *The Design and Implementation of Open ORB 2*, IEEE Distributed Systems Online **2** (2001), no. 6.

[BCF+97] Jérome Besancenot, Michèle Cart, Jean Ferrié, Rachid Guerraoui, Philippe Pucheral, et Bruno Traverson, *Les systèmes transactionnels : concepts, normes et produits*, Collection informatique, Hermès Science, Octobre 1997.

[BCK03] Len Bass, Paul Clements, et Rick Kazman, *Software Architecture in Practice*, 2nd ed., Addison Wesley Professional, Avril 2003.

[BCL+04] Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, et Jean-Bernard Stefani, *An Open Component Model and Its Support in Java*, 7th Int. Symposium on Component-Based Software Engineering (Edinburgh, UK), LNCS, vol. 3054, Springer, Mai 2004, pp. 7–22.

[BCL+06] ———, *The* FRACTAL *component model and its support in Java*, Software : Practice and Experience – Special issue on Experiences with Auto-adaptive and Reconfigurable Systems **36** (2006), no. 11-12, 1257–1284, John Wiley & Sons.

[BCM04] Frédéric Briclet, Christophe Contreras, et Philippe Merle, *Une infrastructure à composants pour le déploiement d'applications à base de composants CORBA*, 1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels (Grenoble, France), IMAG/LSR, Octobre 2004.

[BCS02] Éric Bruneton, Thierry Coupaye, et Jean-Bernard Stefani, *Recursive and Dynamic Software Composition with Sharing*, 7th Int. Workshop on Component-Oriented Programming (Malaga, Spain), Juin 2002.

[BCS04] ———, *The* FRACTAL *Component Model*, France Telecom R&D - INRIA, Grenoble, France, 2.0-3 ed., Février 2004.

[BCSS99] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, et Daniel C. Sturman, *A Case for Message Oriented Middleware*, 13th Int. Symposium Distributed Computing (Bratislava, Slavak Republic), LNCS, vol. 1693, Springer, Septembre 1999, pp. 1–18.

[BDG+94] Alexandros Biliris, Shaul Dar, Narain H. Gehani, H. V. Jagadish, et Krithi Ramamritham, *ASSET : A System for Supporting Extended Transactions*, ACM/SIGMOD Int. Conference on Management of Data (Minneapolis, Minnesota, USA), ACM, Mai 1994, pp. 44–54.

[BJC05] Thaís Vasconcelos Batista, Ackbar Joolia, et Geoff Coulson, *Managing Dynamic Reconfiguration in Component-Based Systems*, 2nd Int. Workshop on Software Architecture (Pisa, Italy), LNCS, vol. 3527, Springer, Juin 2005, pp. 1–17.

[BLLD06] Olivier Barais, Julia Lawall, Anne-Françoise Le Meur, et Laurence Duchien, *Safe Integration of New Concerns in a Software Architecture*, 13th Interntational Conference on Engineering of Computer Based Systems, IEEE, 2006, pp. 52–64.

[BLQ+05] Philippe Bidinger, Matthieu Leclercq, Vivien Quéma, Alan Schmitt, et Jean-Bernard Stefani, *Dream Types - A Domain Specific Type System for Component-Based Message-Oriented Middleware*, 4th Int. ESEC/FSE Workshop on Specification and Verification of Component-Based Systems (Lisbon, Portugal), ACM, Septembre 2005.

[BP95] Roger S. Barga et Calton Pu, *A Practical and Modular Method to Implement Extended Transaction Models*, Int. Conference on Very Large Data Bases (Zurich, Switzerland), 1995, pp. 206–217.

[BP96] ______, *Reflection on a Legacy Transaction Processing Monitor*, Int. Conference on Reflection (San Francisco, CA, USA), Avril 1996.

[BP97] ______, *The Reflective Transaction Framework*, Advanced Transaction Models and Architectures, Kluwer, 1997, pp. 63–89.

[Bri05] Johan Brichau, *Integrative Composition of Program Generators*, PhD dissertation, Vrije Universiteit Brussel, Brussels, Belgium, 2005.

[BVVV05] Martin Bravenboer, Rob Vermaas, Jurgen Vinju, et Eelco Visser, *Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax*, 4th Int. Conference on Generative Programming and Component Engineering (New York, NY, USA), vol. 3676, LNCS, no. 7, Sun Microsystems, Inc., Springer, Octobre 2005, To appear, pp. 157–172.

[CBG+04] Geoff Coulson, Gordon S. Blair, Paul Grace, Ackbar Joolia, Kevin Lee, et Jo Ueyama, *A Component Model for Building Systems Software*, IASTED Int. Conference on Software Engineering and Applications (Cambridge, MA, ESA), Novembre 2004, pp. 1–6.

[CC95] Thierry Coupaye et Christine Collet, *Denotational Semantics for an Active Rule Execution Model*, 2nd Int. Workshop on Rules in Database Systems (London, United Kingdom), LNCS, vol. 985, Springer, 1995, pp. 36–50.

[CCF+05a] Luis Felipe Cabrera, George Copeland, Max Feingold, Robert W Freund, Tom Freund, Jim Johnson, Sean Joyce, Chris Kaler, Johannes Klein, David Langworthy, Mark Little, Anthony Nadalin, Eric Newcomer, David Orchard, Ian Robinson, Tony Storey, et Satish Thatte, *Web Services Atomic Transaction (WS-AtomicTransaction)*, 1.0 ed., Août 2005.

[CCF+05b] Luis Felipe Cabrera, George Copeland, Max Feingold, Robert W Freund, Tom Freund, Jim Johnson, Sean Joyce, Chris Kaler, Johannes Klein, David Langworthy, Mark Little, Anthony Nadalin, Eric Newcomer, David Orchard, Ian Robinson, John Shewchuk, et Tony Storey, *Web Services Coordination (WS-Coordination)*, 1.0 ed., Août 2005.

[CCF+05c] Luis Felipe Cabrera, George Copeland, Max Feingold, Robert W Freund, Tom Freund, Sean Joyce, Johannes Klein, David Langworthy, Mark Little, Frank Leymann, Eric Newcomer, David Orchard, Ian Robinson, Tony Storey, et Satish Thatte, *Web Services Business Activity Framework (WS-BusinessActivity)*, 1.0 ed., Août 2005.

[Che99] Susan Cheung, *Java Transaction Service (JTS)*, Sun Microsystems, Inc., San Antonio Road, Palo Alto, CA, 1.0 ed., Décembre 1999.

[CMRW06] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, et Sanjiva Weerawarana, *Web Services Description Language (WSDL)*, W3C, 2.0 ed., Mars 2006.

[Con06] Denis Conan, *Composition d'entités de contexte de ressources système*, 3ème Conférence Francophone Mobilité et Ubiquité (Paris, France), Septembre 2006.

[CR90] Panayiotis K. Chrysanthis et Krithi Ramamritham, *ACTA : a Framework for Specifying and Reasoning about Transaction Structure and Behavior*, ACM SIGMOD Int. Conference on Management of Data (Atlantic City, New Jersey, USA), Mai 1990, pp. 194–203.

[CR92] Panos K. Chrysanthis et Krithi Ramamritham, *ACTA : the SAGA continues*, Database Transaction Models for Advanced Applications (1992), 349–397, Morgan Kaufmann.

[CR94] ______, *Synthesis of Extended Transaction Models Using ACTA*, ACM Transactions on Database Systems **19** (1994), no. 3, 450–491.

[CSAH98] Panos K. Chrysanthis, George Samaras, et Yousef J. Al-Houmaily, *Recovery Mechanisms in Database Systems*, ch. Recovery and Performance of Atomic Commit Protocols in Distributed and Database Systems, Prentice Hall, 1998.

[DAAC05] Simon Denier, Hervé Albin-Amiot, et Pierre Cointe, *Expression and Composition of Design Patterns with Aspects*, 2ème Journée Francophone sur les Développement de Logiciels Par Aspects (Lille, France), vol. 11, RSTI L'Objet, no. 3, Hermès Science, Septembre 2005.

[Dav05] Pierre-Charles David, *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*, Thèse de doctorat, Université de Nantes, Nantes, France, Juillet 2005.

[DeM03] Linda G. DeMichiel, *Enterprise Java Beans (EJB)*, Santa Clara, California , USA, 2.1 ed., Novembre 2003.

[DEMN98] Roland Ducournau, Jérôme Euzenat, Gérald Masini, et Amedeo Napoli (eds.), *Langages et modèles à objets : état des recherches et perspectives*, Didactique, no. 19, INRIA, Rocquencourt, France, 1998.

[DHL90] Umeshwar Dayal, Meichun Hsu, et Rivka Ladin, *Organizing Long-Running Activities with Triggers and Transactions*, ACM SIGMOD Int. Conference on Management of Data (New York, NY, USA), ACM, 1990, pp. 204–214.

[DHRM05] Christophe Demarey, Gael Harbonnier, Romain Rouvoy, et Philippe Merle, *Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms*, Studia Informatica Universalis Regular Issue **4** (2005), no. 1, 7–24.

[DHTS99] Bruno Dumant, François Horn, F. Dang Tran, et Jean-Bernard Stefani, *Jonathan : An Open Distributed Processing Environment in Java*, Distributed Systems Engineering **6** (1999), no. 1, 3–12, Selected papers from Middleware'98 : The IFIP Int. Conference on Distributed Systems Platforms and Open Distributed Processing.

[DK05] Linda G. DeMichiel et Michael Keith, *Enterprise JavaBeans (EJB) Specification*, Sun Microsystems, Inc., 3.0 ed., 2005.

[DL02] Pierre-Charles David et Thomas Ledoux, *Dynamic Adaptation of Non-Functional Concerns*, ECOOP Workshop on Unanticipated Software Engineering (Malaga, Spain), Juin 2002.

[DL06] ______, *An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components*, 5th Int. ETAPS Symposium on Software Composition (Vienna, Austria), LNCS, vol. 4089, Springer, Mars 2006.

[DM06] Jérémy Dubus et Philippe Merle, *Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués*, 1ère Conférence Francophone sur les Architectures Logicielles (Nantes, France), Hermès Science, Septembre 2006, pp. 13–29.

[Dum06] Cédric Dumoulin, *ModTransf : a model to model transformation engine*, http://www.lifl.fr/west/modtransf, 2006.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, et James O'Toole Jr., *Exokernel : an Operating System Architecture for Application-level Resource Management*, 50th ACM Symposium on Operating Systems Principles (Copper Mountain, Colorado, United States), ACM, 1995, pp. 251–266.

[Esk99] Philip Eskelin, *Component Interaction Patterns*, 6th Annual Conference on the Pattern Languages of Programs (Urbana, IL, USA), Août 1999.

[ESM05] Michael Eichberg, Thorsten Schäfer, et Mira Mezini, *Using Annotations to Check Structural Properties of Classes*, 8th Int. Conference on Fundamental Approaches to Software Engineering (Edinburgh, UK), LNCS, no. 3442, Springer, Avril 2005, pp. 237–252.

[Exe04] François Exertier, *J2EE Deployment : The JOnAS Case Study*, 1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels (Grenoble, France), IMAG/LSR, Octobre 2004.

[Fab05] Johan Fabry, *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*, PhD dissertation, Vrije Universiteit Brussel, Brussels, Belgium, 2005.

[FC05] Johan Fabry et Thomas Cleenewerck, *Aspect-Oriented Domain Specific Languages for Advanced Transaction Management*, 7th Int. Conference on Enterprise Information Systems 2005 (Miami, USA), Mai 2005, Poster session, pp. 428–432.

[FD06a] Johan Fabry et Théo D'Hondt, *A Family of Domain-Specific Aspect Languages on Top of KALA*, 1st AOSD Workshop on Open and Dynamic Aspect Languages (Bonn, Germany), Mars 2006.

[FD06b] ______, *KALA : Kernel Aspect language for advanced transactions*, 21st ACM Symposium on Applied Computing – Track on Programming Languages (Dijon, France), vol. 2, ACM, Avril 2006, pp. 1615–1620.

[FM06] Areski Flissi et Philippe Merle, *A Generic Deployment Framework for Grid Computing and Distributed Applications*, 2nd Int. OTM Symposium on Grid computing, high-performAnce and Distributed Applications (Montpellier, France), Novembre 2006.

[FR03] Marc Fleury et Francisco Reverbel, *The JBoss Extensible Server*, 4th ACM/IFIP/USENIX Int. Middleware Conference (Rio de Janeiro, Brasil), LNCS, vol. 2972, Springer, Juin 2003, pp. 344–373.

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, et Gilles Muller, *Think : A Software Framework for Component-based Operating System Kernels*, USENIX Annual Technical Conference, General Track (Monterey, CA, USA), Juin 2002, pp. 73–86.

[GBS05] Paul Grace, Gordon S. Blair, et Sam Samuel, *A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments*, ACM SIGMOBILE Mobile Computing and Communications Review **9** (2005), no. 1, 2–14, special section on Discovery and Interaction of Mobile Services.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Professional Computing Series, Addison-Westley, USA, 1995.

[GHKM94] Dimitrios Georgakopoulos, Mark F. Hornick, Piotr Krychniak, et Frank Manola, *Specification and Management of Extended Transactions in a Programmable Transaction Environment*, 10th IEEE Int. Conference on Data Engineering (Houston, Texas, USA), IEEE, Février 1994, pp. 462–473.

[GHM+03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, et Henrik Frystyk Nielsen, *SOAP : Messaging Framework*, Microsoft and Sun Microsystems and IBM and Canon, 1.2 ed., Juin 2003, Part 1.

[Gir04] Michael Giroux, *High-Speed ObjectWeb Logger for J2EE Application Servers*, ApacheCon'04, 2004.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, et Dino Mandrioli, *Fundamentals of Software Engineering*, 2 ed., Prentice Hall, Septembre 2002.

[GJSB05] James Gosling, Bill Joy, Guy Steele, et Gilad Bracha, *The Java Language Specification, Third Edition*, Addison-Westley Professional Computing, 2005.

[GMS83] Hector Garcia-Molina et Kenneth Salem, *Using Semantic Knowledge for Transaction Processing in a Distributed Database*, ACM Transactions on Database Systems **8** (1983), no. 2.

[GMS87] ______, *Sagas*, ACM Special Interest Group on Management of Data Annual Conference (San Francisco, California), ACM, Mai 1987.

[GMW00] David Garlan, Robert T. Monroe, et David Wile, *ACME : Architectural Description of Component-Based Systems*, Foundations of Component-Based Systems, Cambridge University Press, 2000, pp. 47–68.

[GR93] Jim Gray et Andreas Reuter, *Transaction Processing : Concepts and Techniques*, Series in Data Management Systems, Morgan Kaufmann, 1993.

[Gra78] Jim Gray, *Notes on Database Operating Systems*, Advanced Course : Operating Systems, LNCS, no. 60, Springer, 1978.

[Gra04] Paul Grace, *Overcoming Middleware Heterogeneity in Mobile Computing Applications*, PhD dissertation, Lancaster University, Lancaster, UK, Mars 2004.

[Hér05] Colombe Hérault, *Adaptabilité des services techniques dans le modèle à composants*, Thèse de doctorat, Université de Valenciennes et du Mont Houy, Valenciennes, France, 2005.

[HHP+05] Daniel Hirschkoff, Tom Hirschowitz, Damien Pous, Alan Schmitt, et Jean-Bernard Stefani, *Component-Oriented Programming with Sharing : Containment is Not Ownership*, 4th Int. Conference on Generative Programming and Component Engineering (Tallinn, Estonia), LNCS, vol. 3676, Springer, Septembre 2005, pp. 389–404.

[HJPP02] Wai Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, et Noël Plouzeau, *A toolkit for weaving aspect oriented UML designs*, of 1st ACM Int. Conference on Aspect Oriented Software Development (Enschede, The Netherlands), ACM, Avril 2002.

[HK02] Jan Hannemann et Gregor Kiczales, *Design Pattern Implementation in Java and AspectJ*, 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (Seattle, Washington, USA), vol. 37, SIGPLAN, no. 11, ACM, Novembre 2002, pp. 161–173.

[HL04] Colombe Hérault et Sylvain Lecomte, *Gestion Dynamique des Services Techniques pour Modele à Composants*, 1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels (Grenoble, France), IMAG/LSR, Octobre 2004, pp. 135–146.

[HMY06] Gang Huang, Hong Mei, et Fu-Qing Yang, *Runtime recovery and manipulation of software architecture of component-based systems*, Automated Software Engineering **13** (2006), no. 2, 257–281.

[HNL04] Colombe Hérault, Sergiy Nemchenko, et Sylvain Lecomte, *A Component-Based Transactional Service, Including Advanced Transactional Models*, 5th Int. Symposium and School on Advance Distributed Systems (Guadalajara, Mexico), LNCS, vol. 3563, Springer, Janvier 2004, pp. 545–556.

[IBM05] IBM Corporation, *Service Component Architecture (SCA) Specification*, 0.9 ed., Novembre 2005.

[JK97] Sushil Jajodia et Larry Kerschberg (eds.), *Advanced Transaction Models and Architectures*, Kluwer, 1997.

[JK05] Frédéric Jouault et Ivan Kurtev, *Transforming Models with ATL*, Int. MoDELS Workshop on Model Transformations in Practice (Montego Bay, Jamaica), LNCS, no. 3844, Springer, Octobre 2005, pp. 128–138.

[Kar03] Randi Karlsen, *An Adaptive Transactional System - Framework and Service Synchronization*, Int. Symposium on Distributed Objects and Applications (Catania, Sicily, Italy), LNCS, vol. 2888, Springer, Novembre 2003, pp. 1208–1225.

[KJ03] Randi Karlsen et Anna-Brith A. Jakobsen, *Transaction Service Management - An approach towards a reflective transaction service*, 2nd Int. Middleware Workshop on Reflective Middleware (Rio de Janeiro, Brasil), PUC-Rio, Juin 2003, pp. 135–138.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, et John Irwin, *Aspect-Oriented Programming*, 11th European Conference on Object-Oriented Programming (Jyväskylä, Finland), LNCS, vol. 1241, Springer, Juillet 1997, pp. 220–242.

[LAA98] M. L. Liu, Divyakant Agrawal, et Amr El Abbadi, *The Performance of Two Phase Commit Protocols in the Presence of Site Failures*, Distributed and Parallel Databases **6** (1998), no. 2, Kluwer.

[LCL06] Marc Léger, Thierry Coupaye, et Thomas Ledoux, *Contrôle dynamique de l'intégrité des communications dans les architectures à composants*, 12ème Conférence Francophone sur les Langages et Modèles à Objets (Nîmes, France), Hermès, Mars 2006, pp. 21–36.

[Lit05] Mark Little, *The Evolution of a Transaction Processing System*, 11th Biennal Workshop on High Performance Transaction Systems (Pacific Grove, California, USA), Septembre 2005.

[LPP+05] Neil Loughran, Nikos Parlavantzas, Monica Pinto, Lidia Fuentes Fernández, Pablo Sánchez, Matthew Webster, et Adrian Colyer, *Survey of Aspect-oriented Middleware*, Survey Deliverable D8, AOSD-Europe-ULANC-10, AOSD-Europe, Juin 2005.

[LQS05] Matthieu Leclercq, Vivien Quéma, et Jean-Bernard Stefani, *DREAM : A Component Framework for Constructing Resource-Aware Configurable Middleware*, IEEE DS Online **6** (2005), no. 9, 1–12.

[LS02] Mark C. Little et Santosh K. Shrivastava, *An Examination of the Transition of the Arjuna Distributed Transaction Processing Software from Research to Products*, 2nd Workshop on Industrial Experiences with Systems Software (Boston, MA, USA), USENIX, Décembre 2002, pp. 41–54.

[LSJ00] Alain Le Guennec, Gerson Sunyé, et Jean-Marc Jézéquel, *Precise modeling of design patterns*, 3rd Int. Conference on the Unified Modeling Language : Advancing the Standard (York, UK), LNCS, vol. 1939, Springer, Octobre 2000, pp. 482–496.

[McI68] M. Douglas McIlroy, *Mass-Produced Software Components*, 1st Int. NATO Conference on Software Engineering (Garmisch Pattenkirchen, Germany), vol. 1, NATO Science Committee, Octobre 1968, pp. 88–98.

[MCWF02] Hong Mei, Feng Chen, Qianxiang Wang, et Yao-Dong Feng, *ABC/ADL : An ADL Supporting Component Composition*, 4th Int. Conference on Formal Engineering Methods (Shanghaï, China), LNCS, vol. 2495, Springer, Octobre 2002, pp. 38–47.

[MDBF06] Raphaël Marvie, Laurence Duchien, et Mireille Blay-Fornarino, *Les plates-formes d'exécution et l'IDM*, ch. 4, p. 236, Hermès Science, Février 2006.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, et Jeff Kramer, *Specifying Distributed Software Architectures*, 5th European Software Engineering Conference (Sitges, Spain), LNCS, vol. 989, Springer, Septembre 1995, pp. 137–153.

[MDK94] Jeff Magee, Naranker Dulay, et Jeff Kramer, *A Constructive Development Environment for Parallel and Distributed Programs*, 2nd IEEE Int. Workshop on Configurable Distributed Systems (Pittsburgh, PA, USA), IEEE Computer Science, Mars 1994, pp. 4–14.

[Mei04] Hong Mei, *ABC : Supporting Software Architectures in the Whole Lifecycle*, 2nd Int. Conference on Software Engineering and Formal Methods (Beijing - China), IEEE, Septembre 2004, pp. 342–343.

[Mes03] Jean-Frédéric Mesnil, *Overview of JOTM : a Java Open Transaction Manager*, 10th Biennal Workshop on High Performance Transaction Systems (Pacific Grove, California, USA), Octobre 2003.

[MFJ05] Pierre-Alain Muller, Frack Fleurey, et Jean-Marc Jézéquel, *Weaving Executability into Object-Oriented Meta-Languages*, 8th Int. Conference on Model Driven Engineering Languages and Systems, LNCS, vol. 3713, Springer, Octobre 2005, Montego Bay, Jamaica, pp. 264–278.

[MH04] Hong Mei et Gang Huang, *PKUAS : An Architecture-Based Reflective Component Operating Platform*, 10th IEEE Int. Workshop on Future Trends of Distributed Computing Systems (Suzhou, China), IEEE, Mai 2004, pp. 163–169.

[MLO86] C. Mohan, B. Lindsay, et R. Obermarck, *Transaction Management in the R* Distributed Database Management System*, ACM Trans. on Database Systems **11** (1986), no. 4.

[Mon01] Robert T. Monroe, *Capturing Software Architecture Design Expertise with Armani*, Rapport technique CMU-CS-98-163, Carnegie Mellon University, Janvier 2001.

[MT00] Nenad Medvidovic et Richard N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering **26** (2000), no. 1, 70–93.

[Nem04] Sergiy Nemchenko, *Modèle de transactions avancées et modèle à composants*, Thèse de doctorat, Université de Valenciennes et du Mont Houy, Valenciennes, France, Septembre 2004.

[Nex05] Next Generation Middleware Group, *OpenCOM : Java versus C++*, Rapport technique, Computing Department – Lancaster University, Lancaster, UK, 2005.

[NNS01] Elie Najm, Abdelkrim Nimour, et Jean-Bernard Stefani, *Behavioural Typing for Objects and Process Calculi*, Formal methods for distributed processing : a survey of object-oriented approaches (2001), 281–301.

[OMG02] OMG, *CORBA Component Model (CCM) Specification*, Needham, MA, USA, 3.0 ed., Septembre 2002.

[OMG03] OMG, *Object Transaction Service (OTS)*, Object Management Group, Needham, MA, USA, 1.4 ed., Septembre 2003.

[OMG04] ______, *Common Object Request Broker Architecture (CORBA)*, Object Management Group, Needham, MA, USA, 3.0.3 ed., Mars 2004, Core Specification.

[OMG05a] OMG, *Additional Structuring Mechanisms for the OTS*, Needham, MA, USA, 1.1 ed., Janvier 2005.

[OMG05b] ______, *Unified Modeling Language (UML) : Superstructure*, Needham, MA, USA, 2.0 ed., Août 2005.

[PAB97] Eric Newcomer Philip A. Bernstein, *Principles of Transaction Processing*, Databases, Morgan Kaufmann, San Francisco, California, USA, Janvier 1997.

[Par72] David L. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM **15** (1972), no. 12, 1053–1058.

[Par05] Nikos Parlavantzas, *Constructing Modifiable Middleware with Component Frameworks*, PhD dissertation, Lancaster University, Lancaster, UK, 2005.

[Paw05] Renaud Pawlak, *Spoon : Annotation-Driven Program Transformation - The AOP Case*, 1st Int. Middleware Workshop on Aspect-Oriented Middleware Development (Grenoble, France), ACM Int. Conference Proceeding Series, vol. 118, ACM, Novembre 2005, pp. 1–6.

[PP01] Marek Procházka et Frantisek Plasil, *Container-Interposed Transactions*, special session on Component-Based Software Engineering of the SNPD international conference (Nagoya, Japan), Août 2001.

[PRC03] Marek Procházka, Romain Rouvoy, et Thierry Coupaye, *On Enhancing Component-Based Middleware with Transactions*, 5th Int. Symposium on Distributed Object and Applications (Catania), LNCS, vol. 2889, Springer, Novembre 2003, Poster session, pp. 1–2.

[Pro01] Marek Procházka, *Advanced Transactions in Enterprise Java Beans*, LNCS (2001), no. 1999, 215.

[Pro02] ______, *Advanced Transactions in Component-Based Software Architectures*, PhD dissertation, Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranské námesti 25, 118 00 Prague 1, Czech Republic, Février 2002.

[PSCD06] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, et Laurence Duchien, *A Model for Developing Component-based and Aspect-oriented Systems*, 5th Int. ETAPS Symposium on Software Composition (Vienna, Austria), LNCS, vol. 4089, Springer, Mars 2006, pp. 259–273.

[PSD$^+$04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, et Laurent Martelli, *JAC : an aspect-based distributed dynamic framework*, Software : Practice and Experience **34** (2004), no. 12, 1119–1148.

[PSWL95] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, et Mark C. Little, *The Design and Implementation of Arjuna*, Computing Systems **8** (1995), no. 3, 255–308, USENIX - MIT Press.

[QBFL04] Vivien Quéma, Roland Balter, André Freyssinet, et Serge Lacourte, *ScalAgent, une plate-forme à composants pour applications asynchrones*, Technique et Science Informatiques **23** (2004), no. 2, 253–274.

[Qué05] Vivien Quéma, *Vers l'exogiciel : une approche de la construction d'infrastructures logicielles radicalement configurables*, Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, Décembre 2005.

[RM03] Romain Rouvoy et Philippe Merle, *Abstraction of Transaction Demarcation in Component-Oriented Platforms*, 4th ACM/IFIP/USENIX Int. Middleware Conference (Rio de Janeiro, Brasil), LNCS, vol. 2972, Springer, Juin 2003, pp. 305–323.

[RM04a] ———, *GoTM : vers un canevas transactionnel à base de composants*, 10ème Conférence sur les Langages, Modèles & Objets (Lille, France), vol. 10, L'Objet, no. 1-3/2004, Hermès Science, Mars 2004, pp. 131–146.

[RM04b] ———, *Towards a Model Driven Approach to Build Component-Based Adaptable Middleware*, 3rd Int. Middleware Workshop on Reflective and Adaptive Middleware (Toronto, Ontario, Canada), ACM Int. Conference Proceeding Series, vol. 80, ACM, Octobre 2004, pp. 195–200.

[RM06a] ———, *Leveraging Component-Oriented Programming with Attribute-Oriented Programming*, 11th Int. ECOOP Workshop on Component-Oriented Programming (Nantes, France), Juillet 2006.

[RM06b] ———, *Using Microcomponents and Design Patterns to Build Evolutionary Transaction Services*, Int. ERCIM Workshop on Software Evolution (Lille, France), Avril 2006.

[RPPM06a] Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, et Philippe Merle, *Apports de la Programmation par Attributs au Modèle de Composants Fractal*, 5èmes Journées Composants (Perpignan, France), Octobre 2006.

[RPPM06b] ———, *Using Attribute-Oriented Programming to Leverage Fractal-Based Developments*, 5th Int. ECOOP Workshop on Fractal Component Model (Nantes, France), Juillet 2006.

[RR00] Jason E. Robbins et David F. Redmiles, *Cognitive support, UML adherence, and XMI interchange in Argo/UML*, Information & Software Technology **42** (2000), no. 2, 79–89.

[RSAM06a] Romain Rouvoy, Patricia Serrano-Alvarado, et Philippe Merle, *A Component-based Approach to Compose Transaction Standards*, 5th Int. ETAPS Symposium on Software Composition (Vienna, Austria), LNCS, vol. 4089, Springer, Mars 2006, pp. 114–130.

[RSAM06b] ———, *Towards Context-Aware Transaction Services*, 6th Int. IFIP Conference on Distributed Applications and Interoperable Systems (Bologna, Italia), LNCS, vol. 4025, Springer, Juin 2006, pp. 272–288.

[SARA04] Patricia Serrano-Alvarado, Claudia Roncancio, et Michel Adiba, *Transactions Adaptables pour les Environnements Mobiles*, Thèse de doctorat, Université Joseph Fourier, Grenoble, France, 2004.

[SARM05] Patricia Serrano-Alvarado, Romain Rouvoy, et Philippe Merle, *Self-Adaptive Component-Based Transaction Commit Management*, 4th Int. Middleware Workshop on Adaptive and Reflective Middleware (Grenoble, France), ACM Int. Conference Proceeding Series, vol. 116, ACM, Novembre 2005, pp. 1–6.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, et Gregory Zelesnik, *Abstractions for Software Architecture and Tools to Support Them*, IEEE Transaction on Software Engineering **21** (1995), no. 4, 314–335.

[SDP91] Santosh K. Shrivastava, Graeme N. Dixon, et Graham D. Parrington, *An Overview of the Arjuna Distributed Programming System*, IEEE Software **8** (1991), no. 1, 66–73.

[SGM02] Clemens Szyperski, Dominik Gruntz, et Stephan Murer, *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley, Novembre 2002.

[SPDC06] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, et Thierry Coupaye, *A Component Model Engineered with Components and Aspects*, 9th Int. SIGSOFT Symposium on Component-Based Software Engineering (Västeras, Sweden), LNCS, vol. 4063, Springer, Juin 2006, pp. 139–153.

[SQ06] Valerio Schiavoni et Vivien Quéma, A Posteriori *Defensive Programming : An Annotation Toolkit for DoS-Resistant Component-Based Architectures*, 21st ACM Symposium on Applied Computing, ACM, 2006.

[Sun02] Sun Microsystems, *Java Management Extensions Instrumentation and Agent (JMX)*, Santa Clara, California, USA, 1.2 ed., Octobre 2002, specification.

[SVV06] Thomas Stahl, Markus Volter, et Bettina Von Stockfleth, *Model-driven Software Development : Technology, Engineering, Management*, John Wiley & Sons, Juillet 2006.

[The92] The Open Group, *Distributed Transaction Processing : The XA Specification*, c193 ed., Février 1992.

[The96] ———, *Distributed Transaction Processing : Reference Model*, 3 ed., Février 1996.

[vKV00] Arie van Deursen, Paul Klint, et Joost Visser, *Domain-Specific Languages : An Annotated Bibliography*, SIGPLAN Notices **35** (2000), no. 6, 26–36, ACM.

[WLD06] Guillaume Waignier, Anne-Françoise Le Meur, et Laurence Duchien, *A Generic Framework for Integrating New Functionnalities into Software Architectures*, 2nd Int. ECOOP Workshop on Architecture-Centric Evolution (Nantes, France), Juillet 2006.

[Wor99] World Wide Web Consortium, *XML Path Language (XPath)*, W3C Recommendation, 1.0 ed., Novembre 1999, http://www.w3.org/TR/xpath.

[WPSO01] Nanbor Wang, Kirthika Parameswaran, Douglas C. Schmidt, et Ossama Othman, *The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware*, 6th USENIX Int. Conference on Object-Oriented Technologies and Systems (San Antonio, Texas, USA), Janvier 2001.

[WR03] Craig Walls et Norman Richards, *XDoclet in Action*, In Actions series, Manning Publications, Décembre 2003.

[WS05] Hiroshi Wada et Junichi Suzuki, *Modeling Turnpike Frontend System : A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming*, 8th Int. Conference on Model Driven Engineering Languages and Systems (Montego Bay, Jamaica), LNCS, vol. 3713, Springer, Octobre 2005, pp. 584 – 600.

[WSTD05a] Hiroshi Wada, Junichi Suzuki, Shingo Takada, et Nohirisa Doi, *A Model Transformation Framework for Domain Specific Languages : An Approach Using UML and Attribute-Oriented Programming*, 9th World Multi-Conference on Systemics, Cybernetics and Informatics (Orlando, FL, USA), Juillet 2005.

[WSTD05b] ———, *Leveraging Metamodeling and Attribute-Oriented Programming to Build a Model-driven Framework for Domain Specific Languages*, 8th JSSST Conference on Systems Programming and its Applications (Gunma, Japan), Japan Society for Software Science and Technology, Mars 2005.

[YWP04] Weihai Yu, Yan Wang, et Calton Pu, *A Dynamic Two-Phase Commit Protocol for Self-Adapting Services*, IEEE Int. Conference on Services Computing (Shanghai, China), IEEE, Septembre 2004, pp. 7–15.