Canevas de développement agile pour l'évolution fiable de systèmes logiciels à composants et orientés services

Guillaume Waignier

▶ To cite this version:

Guillaume Waignier. Canevas de développement agile pour l'évolution fiable de systèmes logiciels à composants et orientés services. Génie logiciel [cs.SE]. Université des Sciences et Technologie de Lille - Lille I, 2010. Français. NNT : . tel-00457590

HAL Id: tel-00457590
https://theses.hal.science/tel-00457590

Submitted on 17 Feb 2010

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

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

Département de formation doctorale en informatique
UFR IEEA

École Doctorale SPI Lille

Canevas de développement agile pour l'évolution fiable de systèmes logiciels à composants et orientés services

THÈSE

présentée et soutenue publiquement le 26 Janvier 2010

pour l'obtention du

Doctorat de l'Université de Lille 1
(spécialité informatique)

par

Guillaume WAIGNIER

Composition du jury :

Président du jury :	M. Gilles GRIMAUD, Professeur	Université Lille 1
Rapporteurs :	M. Antoine BEUGNARD, Professeur	Télécom Bretagne
	M. Philippe LAHIRE, Professeur	Université Nice-Sophia Antipolis
Examinateurs :	Mme. Fabienne BOYER, Maitre de conférences	Université Grenoble I
	M. Nicolas RIVIERRE, Chercheur	Orange Labs
Directeurs de thèse :	Mme Laurence DUCHIEN, Professeur	Université Lille 1
	Mme Anne-Françoise LE MEUR, Maitre de conférences	Université Lille 1

INRIA LILLE - NORD EUROPE - Laboratoire d'Informatique Fondamentale de Lille - UMR CNRS 8022

Résumé

Les systèmes logiciels modernes se caractérisent par un besoin d'évolutions perpétuelles et rapides, comme par exemple dans le monde de l'informatique mobile. Pour faciliter le développement et l'évolution rapide de systèmes complexes, des approches de génie logiciel ont été proposées, telles que les architectures logicielles et la méthode de conception agile. Néanmoins, les solutions actuelles offrent peu de support pour permettre l'évolution fiable d'un système, c'est-à-dire permettre sa modification tout en garantissant le respect de ses exigences de qualités de service et de bon fonctionnement global.

La contribution de cette thèse est CALICO, un canevas de développement agile pour la conception et l'évolution fiable de systèmes logiciels à composants et orientés services. Le développement agile repose sur l'utilisation d'un cycle de développement itératif et incrémental qui permet à l'architecte d'itérer entre les étapes de conception de l'architecture et de débogage du logiciel dans son environnement d'exécution. A chaque itération du cycle, l'architecte peut faire évoluer son logiciel et fiabiliser ses évolutions grâce à l'exécution d'analyses statiques et dynamiques complémentaires. Ainsi, lors de la conception et de l'évolution d'un système, l'architecte dispose d'un ensemble de métamodèles pour spécifier la structure de l'architecture de son logiciel et ses diverses exigences de qualité de services. Lors du déploiement, CALICO utilise les modèles spécifiés pour instancier le système sur la plate-forme d'exécution cible et les garde synchronisés avec le logiciel lors de son exécution. De cette façon, l'architecte dispose toujours d'une vue conceptuelle qui lui permet de raisonner sur les propriétés critiques de son logiciel lors d'une évolution. De plus, pour fiabiliser ces évolutions, CALICO offre un cadre fédérateur qui autorise la réutilisation de nombreux outils d'analyse statique des architectures logicielles et de débogage dynamique qui étaient jusqu'alors dispersés dans différentes plates-formes existantes. Ainsi, chaque évolution peut être analysée statiquement sur la vue conceptuelle avant d'être propagée au système logiciel. Les analyses dynamiques reposent quant à elles sur des valeurs disponibles à l'exécution. La capture de ces valeurs est effectuée grâce à une instrumentation automatique du système logiciel. CALICO permet donc de fiabiliser les évolutions même si les plates-formes d'exécution sous-jacentes ne le proposent pas nativement.

Notre contribution se concrétise par une implémentation multi plates-formes. La version actuelle prend en charge quatre plates-formes à composants et une plate-forme à services. Par ailleurs, les tests de performances que nous avons réalisés démontrent que CALICO est utilisable pour la conception et l'évolution fiable de larges applications jusqu'à 10000 composants et services, ce qui correspond à la montée en charge maximale de la plupart des plates-formes d'exécution.

Mots-clés : Architecture logicielle, développement agile, évolution du logiciel, fiabilisation des évolutions, modèles à l'exécution

Abstract

Modern softwares are characterized by a need for constant and rapid evolution, such as in the mobile domain. To facilitate the development and the rapid evolution of complex systems, software engineering approaches have been proposed, such as software architecture and agile software development. However, current solutions offer poor support to enable the development of a reliable system, i.e, allow its modification while ensuring its compliance with the quality of services requirement and its good overall safety.

The contribution of this PhD thesis is CALICO, an agile development framework for the design and evolution of safe component-based and service-oriented softwares. The agile software development relies on an iterative and incremental development cycle that allows the architect to iterate between the design of the architecture and the debug of the software in its execution context. At each iteration, the architect can evolve its software and check the consistency of its evolution through the execution of static and dynamic analysis tools. Thus, during the design and the evolution of the system, architect can use a set of metamodels to specify the structure of the architecture and its various quality of services requirement. During the deployment, CALICO instantiates the system on the target runtime platform from the models specified and keeps them synchronized with the software during its execution. By this way, the architect still has a conceptual view which allows him to reason on the critical software properties during its evolution. Moreover, in order to check these evolutions, CALICO provides a unifying framework which allows reuse of many static analysis tools of software architectures and dynamic debugging tools, that were scattered in different existing platforms. Thus, each change can be statically analyzed on the conceptual view before being propagated to the software system. Dynamic analysis are based on data values only available during the execution. The capture of these values is done through automatic instrumentation of the software system. Globaly, CALICO enables reliable evolution even if the underlying platforms does not natively provide this support.

Our contribution is concretized by a multi-platform implementation. The current version handles four component-based and service-oriented platforms. Moreover, the benchmarks that we have performed show that CALICO is useable for the design and development of safe applications up to 10,000 components and services, which corresponds to the maximal load of most runtime platforms.

Keyword : Software architecture, agile software development, software evolution, safe evolution, model at runtime

Remerciements

Je tiens à remercier en premier lieu tous les membres du jury qui ont accepté d'évaluer mon travail. Un très grand merci à mes rapporteurs, Antoine Beugnard et Philippe Lahire pour leurs remarques très pertinentes qui m'ont permis d'améliorer la qualité de mon document. Enfin, je tiens à remercier chaleureusement Fabienne Boyer et Nicolas Rivierre d'avoir accepté d'examiner mon travail et Gilles Grimaud d'avoir endossé le rôle de président du jury.

Mes remerciements vont ensuite à Laurence Duchien, ma directrice de thèse et responsable de l'équipe-projet INRIA ADAM. Elle a su me consacrer du temps, même au débotté, malgré son emploi du temps chargé.

Je remercie chaleureusement Anne-Françoise Le Meur pour son encadrement sans faille. Merci pour le logo zinzolin de CALICO que tu as esquissé. J'ai particulièrement apprécié ton grand recul et la rectitude de tes conseils qui m'ont permis de recadrer mes travaux de recherche et de me sortir de maints travaux labyrinthiques. Les moments de ratiocination passés ensemble, ainsi que tes satisfecit et tes critiques m'ont beaucoup aidé. De plus, ta qualité rédactionnelle m'a permis d'éclaircir mes amphigouris et de corriger mes cacographies.

Sans le soutien et les conseils de mes deux encadrantes, cette thèse n'aurait pas pu être ce qu'elle est. Mes trois années de thèse passées avec vous me laisseront un souvenir immarcescible.

Mes remerciements vont ensuite à tous les participants du projet ANR TLog FAROS avec qui j'ai pris beaucoup de plaisir à collaborer.

Je remercie ensuite tous les membres de l'équipe-projet ADAM pour leurs qualités humaines remarquables et leurs encouragements. Vous m'avez accueilli avec aménité et courtoisie.

Je souhaite moulte réussite aux thésards de l'équipe : Carlos Parra, Daniel Romero, Gabriel Hermosillo, Guillaume Libersat, Jonathan Labejof et Russel Nzekwa.

Je tiens à remercier les ingénieurs de l'équipe avec qui j'ai passé de bons moments : Nicolas Dolet, Damien Fournier, Julien Ellart, Nicolas Pessemier, Pierre Carton, Christophe Demarey, Frédéric Loiret, Estéban Duguepéroux, Rémi Melisson, Clément Quinton. Je garderai un souvenir de votre faconde intarissable et de nos tergiversations sans ambages qui animaient les pauses café.

Je ne peux oublier les anciens de l'équipe que j'ai eu le plaisir de côtoyer : Jérémy Dubus, Areski Flissi, Aleš Plšek, Alban Tiberghien, Naouel Moha, Guillaume Dufrêne, Prawee Sriplakich, Carlos Noguera et Angela Lozano. Votre joyeuse alacrité me redonnait des forces pour poursuivre ma thèse jusqu'au bout.

J'associe à ces remerciements Corinne Davoust, Malika Debuysschere, Fatima Hammadi et Maryline Dewaste pour leur gentillesse et leur patience. Sans votre don de l'indispensable viatique je n'aurais pu faire mon chemin.

Enfin, je remercie pour leur soutien mes parents, mon grand frère et mes amis proches. Ils ne m'ont jamais laissé dans la déréliction, ce qui m'a apporté beaucoup de félicité.

Table des matières

1 Introduction **1**
1.1 Problématique . . . 3
1.2 Contexte de travail . . . 5
1.3 Proposition . . . 5
1.4 Organisation du document . . . 7
1.5 Listes des publications liées à cette thèse . . . 7

I État de l'Art **9**

2 Contexte de travail **11**
2.1 Architectures logicielles . . . 11
2.1.1 Architecture à composants . . . 12
2.1.2 Exemple de modèles à composants . . . 13
2.1.3 Architecture orientée services . . . 16
2.1.4 Architecture hybride SCA . . . 17
2.1.5 Bilan . . . 18
2.2 Développement du logiciel . . . 21
2.2.1 Modèles de cycles de développement . . . 22
2.2.2 Etape d'analyse des besoins . . . 24
2.2.3 Etape de conception . . . 25
2.2.4 Etape d'implémentation . . . 30
2.2.5 Etape de validation . . . 33
2.2.6 Etape de déploiement . . . 35
2.3 Conclusion . . . 36

3 Etat de l'art sur l'évolution et la validation d'architectures logicielles **39**
3.1 Gestion de l'évolution dans les architectures logicielles . . . 40
3.1.1 Étape de conception . . . 40
3.1.2 Étape d'implémentation . . . 42
3.1.3 Étape de déploiement . . . 42
3.1.4 Étape d'exécution . . . 43
3.1.5 Bilan . . . 45
3.2 Gestion de la validation statique et dynamique . . . 46
3.2.1 Critères de comparaison . . . 46
3.2.2 Propriétés structurelles . . . 47
3.2.3 Propriétés comportementales . . . 50
3.2.4 Propriétés de flot de données . . . 53
3.2.5 Propriétés de QdS . . . 54

3.2.6 Bilan . . . 56
3.3 Gestion du cycle de vie : le projet FAROS . . . 58
3.3.1 Processus de développement . . . 59
3.3.2 Métamodèle pivot . . . 60
3.3.3 Retour d'expérience . . . 62
3.4 Bilan de l'état de l'art . . . 64
3.4.1 Conclusion . . . 64
3.4.2 Cahier des charges . . . 64

II Contributions **67**

4 CALICO : un cadre générique pour la conception agile d'application **69**
4.1 Motivation . . . 69
4.2 Exemple . . . 70
4.2.1 Scénario d'utilisation . . . 70
4.2.2 Propriétés applicatives du système DMP . . . 71
4.3 Présentation de CALICO . . . 72
4.3.1 Architecture de CALICO . . . 72
4.3.2 Cycle de développement itératif et incrémental . . . 75
4.3.3 Exemple . . . 76
4.4 Organisation de la seconde partie . . . 77

5 Étape de conception du système **79**
5.1 Motivation . . . 79
5.1.1 Spécification de la structure du système . . . 80
5.1.2 Spécification des propriétés applicatives . . . 80
5.2 Spécification de la structure du système . . . 81
5.2.1 Métamodèle de structure du système . . . 81
5.2.2 Gestion des spécificités des plates-formes . . . 84
5.2.3 Exemple . . . 87
5.2.4 Discussion . . . 89
5.3 Spécification des propriétés applicatives . . . 90
5.3.1 Paradigme choisi pour la spécification . . . 90
5.3.2 Spécification structurelle . . . 91
5.3.3 Spécification comportementale . . . 92
5.3.4 Spécification du flot de données . . . 94
5.3.5 Spécification de qualité de service . . . 97
5.3.6 Bilan . . . 98
5.4 Conclusion . . . 98

6 Étape d'analyse du système **101**
6.1 Motivations . . . 101
6.1.1 Cas des interactions partiellement compatibles . . . 102
6.1.2 Extensibilité . . . 103
6.2 Couplage entre les analyses statiques et dynamiques . . . 103
6.2.1 Spécification des analyses dynamiques . . . 103
6.2.2 Spécification des mécanismes d'observation . . . 107
6.3 Analyse de la cohérence du système . . . 110
6.3.1 Composition des spécifications contractuelles . . . 110
6.3.2 Contrat structurel . . . 112
6.3.3 Contrat comportemental . . . 113
6.3.4 Contrat de flot de données . . . 115
6.3.5 Contrat de QdS . . . 118

6.3.6 Intégration des outils d'analyse . . . 121
6.4 Conclusion . . . 123

7 Passage de la conception à la validation dynamique **125**
7.1 Introduction . . . 125
7.1.1 Motivation . . . 126
7.1.2 Organisation de ce chapitre . . . 127
7.1.3 Exemple . . . 128
7.2 Étape d'implémentation . . . 129
7.2.1 Outil de génération de code . . . 129
7.2.2 Algorithme de génération de code . . . 130
7.3 Étape d'instrumentation . . . 132
7.3.1 Principe général . . . 132
7.3.2 Événements structuraux . . . 132
7.3.3 Événements de flot de données . . . 133
7.3.4 Événements de QdS . . . 135
7.4 Étape de déploiement . . . 137
7.4.1 Comparaison des modèles . . . 138
7.4.2 Restructuration des opérations du modèle de mise à jour . . . 141
7.4.3 Interprétation du modèle de mise à jour . . . 147
7.4.4 Discussion . . . 148
7.5 Étape de validation dynamique . . . 149
7.6 Conclusion . . . 150

8 Concrétisation et évaluation de CALICO **153**
8.1 Implémentation de CALICO . . . 153
8.1.1 Implémentation des métamodèles de CALICO . . . 154
8.1.2 Implémentation de l'outillage de CALICO . . . 160
8.1.3 Bilan . . . 165
8.2 Évaluation de CALICO . . . 165
8.2.1 Extensibilité . . . 166
8.2.2 Performance . . . 168
8.2.3 Apport aux acteurs du développement . . . 173
8.3 Conclusion et perspectives . . . 174

9 Conclusion et perspectives **175**
9.1 Résumé des contributions . . . 175
9.2 Perspectives . . . 177
9.2.1 Perspectives à court terme . . . 177
9.2.2 Axes de recherches . . . 178

Bibliographie **190**

Table des figures

2.1 Exemple d'une architecture à composants . . . 13
2.2 Exemple d'une architecture à composants OpenCOM . . . 14
2.3 Exemple d'une architecture à composants Fractal . . . 15
2.4 Exemple d'une architecture à composants CORBA . . . 15
2.5 Exemple d'une architecture SCA . . . 18
2.6 Cycle en cascade . . . 22
2.7 Cycle en V . . . 23
2.8 Cycle itératif et incrémental . . . 24
2.9 Opérateurs de flot de contrôle de BPMN . . . 29

3.1 Boucle de contrôle . . . 44
3.2 Aperçu de FAROS . . . 59
3.3 Métamodèle pivot : structure . . . 60
3.4 Métamodèle pivot : contrat . . . 61
3.5 Métamodèle pivot : événement . . . 63

4.1 Architecture du système DMP . . . 71
4.2 Aperçu de CALICO . . . 74
4.3 Architecture du système DMP après une évolution . . . 77
4.4 Plan de la suite du document . . . 78

5.1 Métamodèle de structure du système . . . 82
5.2 Métamodèle d'intégration . . . 85
5.3 Extrait du modèle de structure du système DMP . . . 88
5.4 Représentation graphique du modèle de la structure du système DMP . . . 88
5.5 Métamodèle des spécifications structurelles . . . 91
5.6 Modèle de la spécification structurelle `S1` . . . 91
5.7 Métamodèle des spécifications comportementales . . . 92
5.8 Modèle de la spécification comportementale `B1` . . . 93
5.9 Représentation graphique des spécifications comportementales `B1`, `B2` et `B4` . . . 94
5.10 Métamodèle des spécifications de flot de données . . . 95
5.11 Réprésentation graphique des spécifications de flot de données `D1` à `D4` . . . 96
5.12 Modèle de la spécification de flot de données `D1` . . . 96
5.13 Métamodèle des spécifications de QdS . . . 97
5.14 Modèle de la spécification de QdS `QdS1` . . . 98

6.1 Métamodèle de débogage . . . 104
6.2 Métamodèle de FAC . . . 108
6.3 Métamodèle d'aspect de CALICO . . . 109
6.4 Principe global d'une analyse statique . . . 111

6.5 Métamodèle des contrats structurels . . . 112
6.6 Métamodèle des contrats comportementaux . . . 113
6.7 Schéma du comportement de l'assemblage . . . 115
6.8 Métamodèle des contrats de flot de données . . . 116
6.9 Schéma de composition des spécifications de flot de données dans l'assemblage . . 117
6.10 Analyse dynamique de la spécification de flot de données `D1` . . . 118
6.11 Métamodèle des contrats de QdS . . . 119
6.12 Métamodèle d'intégration des outils d'analyse . . . 121

7.1 Organisation des outils du support d'exécution de CALICO . . . 127
7.2 Première version de l'architecture du système DMP . . . 128
7.3 Seconde version de l'architecture du système DMP . . . 129
7.4 Métamodèle d'intégration . . . 129
7.5 Aspect associé à l'analyse dynamique du flot de données . . . 134
7.6 Métamodèle d'intégration . . . 136
7.7 Organisation de l'outil de chargement . . . 137
7.8 Métamodèle de mise à jour . . . 139
7.9 Architecture interne de l'étape d'ajustement . . . 142
7.10 Gestion des intercepteurs . . . 145
7.11 Exemple d'organisation interne . . . 146
7.12 Métamodèle d'intégration . . . 147
7.13 Métamodèle des spécifications contractuelles . . . 149

8.1 Métamodèle simplifié d'EMF . . . 154
8.2 Capture d'écran de l'éditeur arborescent . . . 155
8.3 Chaîne de processus de génération d'EMF . . . 156
8.4 Chaîne de processus de génération de GMF . . . 157
8.5 Modeleur graphique structurel de CALICO . . . 158
8.6 Modeleur graphique comportemental de CALICO . . . 159
8.7 Modeleur graphique des points de communication de CALICO . . . 159
8.8 Interface `IAnalyzer` . . . 160
8.9 Interface `ICodeManipulation` . . . 162
8.10 Interface `IPlatFormSensor` . . . 163
8.11 Interface `IFilter` . . . 164
8.12 Interface `IPlatformDriver` . . . 164
8.13 Interface `ISensor` . . . 164
8.14 Interface `IAction` . . . 165
8.15 Extension de CALICO . . . 166
8.16 Architecture du système de test . . . 168
8.17 Durée d'analyse du système . . . 169
8.18 Durée du déploiement de tous le système . . . 170
8.19 Temps mis pour déployer un nouveau composant dans le système . . . 170
8.20 Consommation mémoire de CALICO . . . 171
8.21 Architecture du système *"pipe and filter"* . . . 171
8.22 Durée de la réification des événements . . . 172

Liste des tableaux

2.1 Comparaison des modèles à composants et des modèles à services 19
2.2 Annotations Fractal 31
2.3 Annotation pour les services WEB 32

3.1 Support des propriétés applicatives dans les modèles à composants et à services . 57

6.1 Opérateurs de composition des spécifications structurelles 112
6.2 Opérateurs de composition des spécifications comportementales 114
6.3 Opérateurs de composition des spécifications de flot de données 116
6.4 Exemple d'opérateurs de composition de la QdS (temps de réponse maximal) . . . 119
6.5 Exemple d'opérateurs de composition de la QdS (temps de réponse minimal) . . . 119

7.1 Interprétation du modèle de mise à jour 148

8.1 Nombre de lignes de code pour le modeleur graphique 158
8.2 Nombre de lignes de code pour l'outil d'analyse 161
8.3 Nombre de lignes de code pour l'outil de génération de code et de chargement . . 162
8.4 Nombre de lignes de code pour l'outil d'instrumentation 163

Liste des listings

2.1 Extrait de description Fractal . . . 26
2.2 Extrait de description CCM . . . 26
2.3 Extrait de description SCA . . . 27
2.4 Extrait de description WSDL . . . 27
2.5 Extrait de description BPEL . . . 28
2.6 Implémentation Fractal du composant `Client` . . . 30
2.7 Implémentation Fractal du composant `Client` avec les annotations . . . 32
3.1 Patron d'architecture logicielle de cache . . . 41
3.2 Exemple d'aspect . . . 42
3.3 Exemple de règle de reconfiguration pour Plastik . . . 45
3.4 Exemple de contrainte OCL . . . 47
3.5 Exemple de contrainte CCLJ . . . 48
3.6 Exemple de contrainte Armani . . . 49
3.7 Exemple de spécification SFSP . . . 51
5.1 Attributs OCL portant sur le métamodèle de la structure du système . . . 83
5.2 Contraintes OCL portant sur le métamodèle de la structure du système . . . 83
5.3 Profil OpenCOM . . . 85
5.4 Profil Fractal . . . 86
5.5 Profil OpenCCM . . . 86
5.6 Profil FraSCAti . . . 87
5.7 Profil des services Web . . . 87
6.1 Grammaire du langage de coupe de FAC . . . 107
6.2 Contraintes OCL du métamodèle d'intégration . . . 121
7.1 Squelette de code de l'entité `DataConverter` . . . 130
7.2 Extrait de l'ADL du système DMP . . . 131
7.3 Instrumentalisation de l'entité `DataConverter` . . . 134
7.4 Contraintes OCL du métamodèle d'intégration . . . 136

Chapitre 1

Introduction

Sommaire

1.1 Problématique 3
1.2 Contexte de travail 5
1.3 Proposition 5
1.4 Organisation du document 7
1.5 Listes des publications liées à cette thèse 7

Les systèmes logiciels modernes se distinguent par un besoin d'*évolution* rapide et une *complexité* croissante, avec notamment l'apparition des nouveaux domaines d'applications, comme par exemple les logiciels destinés aux périphériques mobiles, c'est-à-dire les nouveaux téléphones et PDAs. En effet, dans ces domaines, les utilisateurs réclament que les logiciels fournissent toujours plus de fonctionnalités et les logiciels ont donc besoin de prendre en compte ces nouvelles exigences demandées. De plus, en raison de la très grande hétérogénéité des périphériques mobiles et des avancées rapides des nouvelles technologies matérielles, il est aussi nécessaire d'adapter les logiciels en fonction de l'infrastructure matérielle et du système d'exploitation. De plus, dans la mesure où le logiciel occupe une place omniprésente dans la société, les utilisateurs exigent que le logiciel soit *fiable* et qu'il satisfasse un certain niveau de qualité de services. Par exemple, un disfonctionnement d'un logiciel sur un téléphone peut avoir des répercussions sur les fonctionnalités du téléphone en lui même, telles qu'une incapacité du téléphone à recevoir et émettre des appels.

Il est donc nécessaire de permettre l'évolution fiable des logiciels complexes.

Evolution du logiciel. Lehman a défini la notion d'évolution du logiciel ainsi :

"Continued satisfaction demands continuing changes. The system will have to be adapted to a changing environment, changing needs, developing concepts and advancing technonolgies. The application and the system should evolve." [LB85]

Dans cette définition, l'évolution d'un logiciel consiste à adapter le logiciel afin qu'il respecte les nouvelles exigences des utilisateurs et qu'il suive les modifications de l'environnement et les avancées technologiques. Dans le cadre de cette thèse, nous considérons que l'évolution correspond à adapter le logiciel en ajoutant de nouvelles fonctionnalités et en supprimant les fonctionnalités obsolètes.

Afin de répondre à ce besoin d'évolution du logiciel, les méthodes de conception agiles ont émergé et ont été officialisées en 2001 avec la rédaction du manifeste Agile [All01]. Ces méthodes permettent une plus grande réactivité vis-à-vis de l'évolution du besoin des clients. La méthode de développement agile repose sur l'utilisation d'un cycle de développement itératif

et incrémental [LB03]. Ainsi, avec cette méthode, le logiciel est élaboré en itérant les étapes d'analyse des besoins des utilisateurs, de conception, d'implémentation et de validation dynamique du logiciel. Chaque itération du cycle ajoute un incrément dans le logiciel, c'est-à-dire qu'elle ajoute de nouvelles fonctionnalités ou supprime les fonctionnalités obsolètes. Durant l'étape d'analyse des besoins, l'analyste établit le cahier des charges fonctionnelles et techniques qui détaille les fonctionnalités que doit offrir le logiciel, ainsi que les contraintes auxquelles il est soumis. Pendant l'étape de conception, l'architecte logiciel décrit, à un haut niveau d'abstraction, l'architecture logicielle de son système, c'est-à-dire sans se préoccuper des détails d'implémentation. Pendant l'étape d'implémentation, les développeurs écrivent le code du logiciel conformément aux spécifications de l'architecte. Enfin, pendant la phase de validation dynamique, les testeurs expérimentent, dans un contexte d'exécution réel, les fonctionnalités du système. Ils doivent, pour cela, déployer et installer le système sur la ou les machines cibles, puis exécuter les scénarios d'utilisation du logiciel. Dans le cas où le comportement du logiciel à l'exécution n'est pas conforme à celui attendu, les testeurs doivent faire remonter l'erreur à l'architecte logiciel. L'architecte peut ainsi corriger son logiciel dans l'itération suivante du cycle de développement en recommençant à partir de l'étape de conception.

Logiciels complexes. Le développement de logiciels complexes est une tâche ardue. Le génie logiciel propose deux grands principes pour l'élaboration de tels systèmes : la modularisation et la réutilisation [GJM91]. La modularisation vise à encapsuler et isoler, le plus possible, chaque fonctionnalité dans des briques logicielles différentes. Le logiciel résultant est donc vu comme un assemblage de briques qui fournissent dans leur ensemble toutes les fonctionnalités attendues par les utilisateurs. Un bon découpage modulaire se caractérise par une forte cohésion interne des briques logicielles et un faible couplage entre les briques. Ce principe rend possible le développement de chaque brique logicielle par des équipes différentes. La réutilisation consiste à autoriser l'utilisation des mêmes briques logicielles pour concevoir différents logiciels. Ce principe permet donc d'augmenter la vitesse de développement du logiciel.

Une application de ces deux principes a donné lieu au paradigme d'architecture logicielle [MT00]. Avec ce paradigme, un logiciel est décrit par son architecture. L'architecture spécifie les briques logicielles qui composent l'application, ainsi que les interactions entre ces briques. Il existe deux grandes familles d'architecture logicielle : les architectures à composants [Szy97] et les architectures orientées services [Erl05, OAS08]. Dans le cas des architectures à composants, la brique logicielle est appelée un composant. Chaque composant logiciel explicite les fonctionnalités qu'il offre et qu'il requiert au travers de ses interfaces. Un composant logiciel encapsule un ensemble de fonctionnalités et permet alors une réutilisation de ces dernières en intégrant le composant logiciel dans le système final. Fondamentalement, l'architecture à composants permet un découpage structurel du système. Dans le cas des architectures orientées services, la brique logicielle est appelée service. Le service décrit explicitement la fonctionnalité qu'il offre. A la différence des architectures à composants, la vision de l'architecture logicielle n'est pas structurelle mais comportementale, c'est-à-dire que l'architecture orientée services exprime l'enchaînement des appels vers d'autres services nécessaires pour remplir l'application souhaitée. Les services mettent l'accent sur la réutilisation et l'interopérabilité, ils permettent à chaque entreprise d'exporter leurs fonctionnalités métiers, comme par exemple un service météo ou un service de réservation de billets de train.

Les architectures à composants et orientées services ont donné lieu à une multitude de modèles et de plates-formes d'exécution différentes. Du coté des architectures à composants, de nombreux modèles à composants ont été définis par des consortiums industriels ou des laboratoires académiques. Par exemple le modèle à composants Fractal [BCS04] a été élaboré en collaboration avec le laboratoire de recherche INRIA et France Télécom R&D. Le modèle à composants CORBA a été spécifié par l'OMG (*"Object Management Group"*) [Obj02b], plus récemment les modèles à composants EJB (*"Enterprise JavaBeans"*) [SUN06b], *.net* [Low05] et OSGi [OSG07] ont été élaborés, respectivement, par SUN Microsystems, Microsoft et l'alliance OSGi. Pour le moment, aucun modèle n'a acquis de prédominance par rapport aux autres. Il existe donc une

très grande hétérogénéité de modèles et de plates-formes à composants qui coexistent. Du coté des architectures orientées services, les services WEB sont les plus représentatifs. Les services WEB ont été définis par des sociétés de normalisation de l'internet, comme le W3C [W3C01]. Le fondement des services WEB repose sur l'utilisation des standards de l'internet et du WEB pour établir la communication entre les différents services WEB. Actuellement, les différents modèles d'architectures à composants et d'architectures orientées services co-existent. Les entreprises ont donc besoin de faire interopérer ces différents modèles. Pour répondre à ce besoin, le consortium OASIS a défini le modèle SCA (*"Service Component Architecture"*) [BII+07] qui est un modèle hybride qui réunit différents modèles à composants et les services WEB.

Logiciels fiables. L'ISO 9126 définit la fiabilité du logiciel comme étant l'aptitude du logiciel à maintenir son niveau de service dans des conditions déterminées pendant une période de temps définie [ISO03]. Dans le cadre de cette thèse, nous considérons que les composants et les services sont conformes à leurs spécifications, c'est-à-dire qu'ils fournissent les fonctionnalités décrites. Notre travail se focalise sur la compatibilité des interactions entre les composants et les services.

Dans cet objectif, la plupart des modèles à composants et des modèles orientées services offrent des outils qui permettent de vérifier que l'architecture décrite respecte les contraintes d'assemblage du modèle [BCS04, Obj02b]. Cependant, cette vérification est principalement structurelle et repose sur la compatibilité de type entre les composants et les services qui interagissent entre eux.

Pour concevoir un logiciel fiable, il est aussi nécessaire de prendre en compte les exigences qui sont liées au métier de l'application. Ces exigences peuvent concerner les relations structurelles et comportementales entre les différents composants ou services [Hoa85, Mag99, Mon01]. Elles peuvent aussi prendre en compte les propriétés de qualité de services du logiciel, comme la performance ou la sécurité du logiciel [IBM03], ainsi que les propriétés de flot de données qui contraignent la valeur des données échangées entre les composants ou les services [CR99, JRMWC07]. Une fois que les exigences applicatives de chaque composant ou service sont décrites, il est important de déterminer si la composition de composants et de services constituant l'architecture satisfait les exigences. Comme les systèmes logiciels sont complexes, il n'est pas envisageable pour l'architecte d'analyser manuellement son système. Il a donc besoin d'outils d'analyse. Dans cet objectif, quelques modèles autorisent les architectures logicielles à spécifier et vérifier les diverses exigences que le logiciel doit satisfaire, comme par exemple Darwin [MDEK95], Wright[All97] et ConFract [CR99]. Ces exigences sont généralement décrites à l'aide de propriétés applicatives qui sont attachées sur les composants et les services constituant le logiciel. Elles peuvent être vérifiées avec les outils d'analyses statiques ou dynamiques associées à ces modèles. Les outils d'analyse statique vérifient que l'architecture spécifiée ne viole pas les propriétés applicatives sans exécuter le logiciel [MDEK95, All97]. Les outils d'analyse dynamique testent si les propriétés applicatives sont violées en exécutant le logiciel. Néanmoins, les modèles qui offrent un support pour spécifier et analyser les propriétés applicatives restent minoritaires. De plus, ce support est très disparate selon les modèles à composants et orientées services, aucun modèle permet la spécification et la vérification de l'ensemble des exigences décrites précédemment.

1.1 Problématique

Dans le cadre de cette thèse, nous traitons la problématique de l'évolution fiable des applications à composants et orientées services. Nous avons choisi de prendre en compte, à la fois les applications à composants et les applications orientées services, car ces deux grandes familles coexistent dans le monde industriel, et donc les entreprises ont besoin de faire évoluer ces deux grandes familles de système. Nous avons décomposé cette problématique en trois problèmes qui sont : (1) permettre le développement agile pour construire les systèmes à composants ou orientés services avec un cycle de développement itératif et incrémental, (2) fiabiliser les évolutions et (3) surmonter le manque de fonctionnalités de validation statique et dynamique

des différentes plates-formes d'exécution.

(1) Les canevas de développement qui existent ne sont pas adaptés pour faire évoluer et mettre au point un logiciel à composants ou orienté services. En effet, d'une part, bien que la méthode de conception agile soit efficace pour élaborer rapidement un logiciel qui suit l'évolution des besoins des utilisateurs, cette méthode est malheureusement très peu utilisée dans le cadre du développement de logiciel à composants ou orienté services [RG03]. Au contraire, le développement de système logiciel à composants ou orienté services repose sur un cycle de développement descendant, comme le cycle de développement en cascade [Roy70] ou le cycle en V [IAB97] qui ne sont pas bien adaptés pour faire évoluer des systèmes logiciels.

D'autre part, pour faire évoluer un logiciel ou le mettre au point, il est préférable de modifier uniquement les composants ou les services qui sont concernés par l'évolution. De cette manière, les fonctionnalités qui ne sont pas concernées par l'évolution sont toujours disponibles pour les utilisateurs pendant l'évolution du logiciel. Or, les canevas de développement existants ne prennent pas en compte cette problématique. En effet, le support de reconfiguration dynamiquement du logiciel en cours d'exécution, c'est-à-dire ajouter et supprimer des composants/services [BCL+04, BCM04, OSG07, SMF+09], n'est pas adapté pour concevoir un logiciel. Avec cette approche, la vue conceptuelle de l'architecture requise par l'architecte pour faire évoluer son logiciel est perdue. Concrètement, l'architecte ne peut pas concevoir et faire évoluer son architecture de manière uniforme et transparante.

(2) Les approches actuelles ne permettent pas de s'assurer que les évolutions sont fiables. En effet, lors d'une évolution d'un logiciel, il existe deux sources d'incohérences potentielles. D'une part, des incohérences peuvent être introduites entre chaque étape du cycle de développement. En effet, les canevas de développement actuels brisent la continuité du cycle de développement du logiciel. Chaque transition entre les étapes du cycle est effectuée manuellement, c'est-à-dire que le passage de la conception à l'implémentation et le passage de l'implémentation au déploiement, puis à l'exécution du système mettent en jeu des actions humaines avec des abstractions logicielles différentes. En conséquence, ces tâches manuelles peuvent potentiellement introduire des incohérences entre l'architecture décrite par l'architecte et le système en cours d'exécution. Il est donc nécessaire de fournir aux différents acteurs, c'est-à-dire l'architecte, les développeurs et les testeurs, un moyen pour construire le logiciel qui limite l'introduction d'incohérence lors du passage entre les étapes du cycle de développement.

D'autre part, après une évolution, le logiciel résultant peut violer les propriétés applicatives. En effet, une évolution modifie la composition des composants et des services qui forment le logiciel. Or, la plus grande partie des canevas existants n'effectuent pas de vérification après chaque évolution, comme par exemple [GCH+04, Sic08, RBD+09]. De plus, les quelques canevas qui effectuent des analyses après une évolution ne permettent pas à l'architecte de corriger sa conception si un problème est détecté, comme par exemple [KRCQM08].

Mais aussi, les outils d'analyse existants qui vérifient si l'architecture du logiciel viole des propriétés applicatives ne sont pas complètement satisfaisants : ils effectuent soit uniquement des analyses statiques [Mon01, Bar05], soit uniquement des analyses dynamiques [IBM03, JRMWC07]. Donc, ces approches manquent de précisions et de finesses pour analyser la violation des propriétés qui dépendent de données qui ne sont connues que lors de l'exécution du système, comme par exemple l'analyse d'une contrainte sur le temps de réponse d'un service. En effet, les analyses purement statiques ne sont pas capables d'identifier un problème provenant de données d'exécution. Dans le cas des analyses dynamiques, les outils insèrent des assertions exécutables dans le code de l'application, sans prendre en compte l'architecture globale du logiciel. Donc, sans avoir eu recours à une analyse statique, ces approches purement dynamiques ne sont pas capables d'identifier si elles ont ajouté des assertions qui seront soit toujours fausses, soit toujours vraies. Elles peuvent donc potentiellement insérer des tests inutiles dans le code du système qui alourdissent l'exécution du logiciel. La raison de cette lacune au niveau des outils d'analyse vient du fait qu'il existe une dissociation entre les analyses statiques et dynamiques qui sont des approches complémentaires.

(3) Les fonctionnalités disponibles pour analyser une architecture sont très disparates selon la plate-forme d'exécution cible. La majorité des plates-formes à composants ou orientées services manquent d'outils pour analyser statiquement et dynamiquement le système logiciel, comme par exemple les plates-formes à composants OpenCCM [BCM04] et OpenCOM [CBG+04], la plate-forme orientée services GlassFish [SUN09] et les plates-formes hybrides SCA FraSCAti [SMF+09] et Tuscany [Apa09]. De plus, dans les cas où la plate-forme à composants ou orientée services offre ces outils, ces derniers sont fortement couplés avec elle. Par exemple, si la plate-forme cible ne fournit pas l'analyse désirée, l'architecte logiciel n'a pas d'autre choix que de retranscrire la description de son système dans une autre plate-forme à composants ou orientée services qui gère l'analyse. Ces lacunes au niveau des plates-formes limitent grandement la possibilité de fiabiliser les évolutions.

1.2 Contexte de travail

Ce travail de thèse a été financé avec une bourse ministérielle et s'est déroulé au sein de l'équipe-projet ADAM. Cette équipe est sous la tutelle de l'INRIA, du CNRS (UMR 8022) avec le Laboratoire d'Informatique Fondamentale de Lille (LIFL) et de l'Université de Lille 1. L'équipe-projet ADAM est une équipe de recherche en génie logiciel qui se focalise sur le challenge de l'*adaptation* des logiciels. Le terme adaptation doit être pris au sens large, c'est-à-dire qu'il n'inclut pas seulement l'adaptation du logiciel pendant son exécution, mais prend aussi en compte l'évolution du logiciel par ajout ou suppression de fonctionnalités en concordance avec l'évolution des besoins des utilisateurs. Pour l'équipe-projet ADAM, l'adaptation est un problème de première classe et il doit être pris en compte durant tout le cycle de vie du logiciel. L'objectif de l'équipe est de fournir un ensemble de paradigmes, d'approches et de canevas logiciels reposant sur des techniques d'ingénierie logicielle avancées, comme l'ingénierie à composants ou le développement orienté aspect, afin de construire des logiciels adaptables distribués et de prendre en compte l'adaptation durant tout le cycle de vie du logiciel, de la conception à l'exécution.

Durant ma thèse, j'ai participé au projet ANR RNTL FAROS [1] dont l'objectif était de définir un environnement de composition pour la construction fiable d'architectures orientées services. La démarche adoptée a consisté à prendre en compte des éléments contractuels permettant d'offrir des garanties lors des compositions de services et de mettre au point un procédé de construction allant des modèles métiers jusqu'à leur projection vers des plates-formes d'exécution.

1.3 Proposition

Dans le cadre de cette thèse, nous proposons de reveler le challenge de l'évolution fiable du logiciel. Nous répondons aux trois problématiques levées précédemment, à savoir, (1) permettre le développement agile pour construire les systèmes à composants ou orientés services avec un cycle de développement itératif et incrémental, (2) fiabiliser les évolutions et (3) surmonter le manque de fonctionnalités de validation statique et dynamique des différentes plates-formes d'exécution.

Nous proposons donc une approche de développement et un canevas logiciel associé permettant un développement agile et fiable de grandes applications. Notre canevas se nomme CALICO pour *"Component AssembLy Interaction Control FramewOrk"*. CALICO autorise le développement d'applications à composants ou orientées services en utilisant un cycle de développement itératif et incrémental. CALICO est constitué de deux niveaux que nous appelons le *niveau modèle* et le *niveau plate-forme*. Le niveau modèle fournit un ensemble de métamodèles qui permettent à un architecte de spécifier la structure de son architecture, ainsi que les diverses propriétés applicatives, c'est-à-dire les propriétés structurelles, comportementales, de flot de données et de

[1] http ://www.lifl.fr/faros

qualité de services. Le niveau plate-forme contient le système logiciel en cours d'exécution sur une plate-forme d'exécution. Concrètement, à partir des modèles de l'architecture spécifiés, CALICO déploie le logiciel dans la plate-forme d'exécution choisie. Ces modèles sont constamment gardés synchronisés avec le logiciel en cours d'exécution de telle sorte que l'architecte dispose toujours d'une vue conceptuelle de son architecture, ce qui lui permet de raisonner et de faire évoluer son logiciel. Afin de fiabiliser les évolutions, CALICO propose un cadre fédérateur qui autorise la réutilisation des outils d'analyse statique et dynamique existants. Ainsi, les évolutions décrites par l'architecte sont propagées dans la plate-forme d'exécution cible uniquement si elles ne violent pas les propriétés applicatives du logiciel. CALICO est donc capable d'appliquer les évolutions de conception effectuées lors de chaque itération du développement sans arrêter l'exécution du logiciel, afin de garder le logiciel disponible.

CALICO couple très fortement les différentes étapes du cycle de développement, ce qui limite l'introduction d'incohérence lors des transitions entre les différentes étapes du cycle. De plus, ce couplage permet aussi de faire le lien entre les analyses statiques et dynamiques. Ainsi, après chaque évolution, les analyses statiques vérifient que les propriétés applicatives ne sont pas violées. Lorsque les analyses statiques ont besoin de données seulement disponibles lors de l'exécution du logiciel pour finaliser la validation, comme par exemple un temps de réponse ou la valeur d'un message échangé, alors CALICO détermine automatiquement les analyses dynamiques qui devront être effectuées pendant l'étape de validation dynamique. CALICO insère aussi automatiquement dans le logiciel, le mécanisme nécessaire pour capturer les données d'exécution requises par les analyses dynamiques. Ainsi, si les analyses dynamiques détectent une erreur, CALICO pointe à l'architecte l'élément de conception qui est la source de l'erreur de telle sorte que l'architecte peut corriger directement le problème en modifiant son architecture au niveau modèle. La modification de la description déclenche les analyses statiques, puis la propagation des changements sur l'application en cours d'exécution si les analyses n'ont identifié aucune violation des propriétés applicatives. L'architecte peut itérer ce processus jusqu'à l'obtention d'un système fiable.

Face aux trois problématiques, CALICO apporte les contributions principales suivantes :

- Un canevas de conception agile pour le développement incrémental et itératif de logiciels à composants ou orientés services [2]. Ce canevas repose sur une démarche d'ingénierie dirigée par les modèles [EVI05]. Ainsi, le niveau modèle de CALICO contient un ensemble de métamodèles qui permet à l'architecte logiciel de concevoir son architecture logicielle.
- Une fiabilisation des évolutions. En effet, lorsqu'une évolution viole des propriétés applicatives, CALICO permet à l'architecte de corriger sa conception avant de propager cette évolution dans le système. Ce mécanisme se base, d'une part, sur un couplage fort entre les étapes du cycle de développement, c'est-à-dire la conception, l'implémentation, le déploiement et la validation dynamique, afin d'éviter toute introduction d'incohérence entre les étapes du cycle. D'autre part, il repose sur un couplage fort entre les analyses statiques et dynamiques.
- Un canevas multi plates-formes. Dans CALICO, le support spécifique à chaque plate-forme cible, comme la génération du squelette code ou la propagation des évolutions du niveau modèle au niveau plate-forme, est clairement séparé du support générique. De plus, les outils d'analyse dispersés dans les différentes plates-formes à composants et orientées services sont réutilisables dans CALICO, ce qui permet d'exploiter ces outils sur des plates-formes qui n'en offrent pas de base. En outre, CALICO résout le manque de fonctionnalités liées aux analyses dynamiques offertes par les plates-formes en intégrant automatiquement ces fonctionnalités dans le logiciel. Ce mécanisme repose sur des mécanismes de tissage d'aspects [KLM+97], de génération de code et sur l'intégration de canevas de sondes [DL05].

[2] http ://calico.gforge.inria.fr

1.4 Organisation du document

La suite du document est composée de huit chapitres regroupés dans deux parties. La partie I concerne l'état de l'art autour des travaux liés au développement de logiciels à composants ou orientés services, à leur évolution et leur fiabilisation.

Chapitre 2. Ce chapitre donne un bref aperçu des concepts présents dans les architectures logicielles et décrit les modèles à composants et orientés services. Ensuite, il détaille les différents modèles de cycle de développement d'un logiciel et explique les différentes approches qui entrent en jeu dans chacune des étapes du cycle.

Chapitre 3. Ce chapitre présente les travaux relatifs à l'évolution et à la fiabilisation de logiciels à composants ou orientés services. D'abord, il donne un aperçu des travaux existants portant sur l'évolution du logiciel pour chacune des étapes du cycle de développement d'une application à composants ou orientée services. Ensuite, il présente différents travaux concernant la spécification et l'analyse des propriétés applicatives dans une architecture à composants ou orientée services et détermine les points forts et les faiblesses des outils et travaux existants. Pour finir, ce chapitre dresse le cahier des charges listant les fonctionnalités qui sont requises pour produire un canevas de développement agile pour la conception et l'évolution d'applications à composants ou orientées services fiables.

La partie II décrit notre proposition CALICO : un canevas pour le développement agile et fiable d'architecture à composants ou orientée services.

Chapitre 4. Ce chapitre donne un aperçu global de notre canevas logiciel CALICO.

Chapitre 5. Ce chapitre explique comment un architecte logiciel conçoit son architecture dans CALICO. Les métamodèles de CALICO permettant l'expression de la structure et des propriétés structurelles, comportementales, de flot de données et de qualité de services y sont présentés.

Chapitre 6. Ce chapitre décrit le support générique des outils d'analyse statique du système et présente son utilisation pour l'intégration d'outils d'analyse structurelle, comportementale, de flot de données et qualité de services.

Chapitre 7. Ce chapitre présente le mécanisme qui rend possible la propagation des évolutions décrites au niveau modèle dans le système en cours d'exécution au niveau plate-forme. Il détaille l'étape d'implémentation, de déploiement et de validation dynamique du logiciel avec CALICO, puis il explique comment le couplage fort entre les étapes de conception, d'implémentation et de validation dynamique permet un développement itératif.

Chapitre 8. Ce chapitre regroupe les détails sur l'implémentation de CALICO, l'évaluation des performances du canevas et son passage à l'échelle.

Enfin le chapitre 9 conclut ce document et présente quelques perspectives associées à ce travail.

1.5 Listes des publications liées à cette thèse

Conférences internationales

- **A Model-Based Framework to Design and Debug Safe Component-Based Autonomic Systems.** Guillaume Waignier, Anne-Françoise Le Meur and Laurence Duchien. In *Proceedings of the 5th International Conference on the Quality of Software-Architectures* (QoSA 2009), Pennsylvania, USA, jun 2009. (Rank Core A. Taux d'acceptation=39%)
- **Architectural Specification and Static Analyses of Contractual Application Properties.** Guillaume Waignier, Anne-Françoise Le Meur and Laurence Duchien. In *Proceedings of the 4th International Conference on the Quality of Software-Architectures* (QoSA 2008), pages 152-170, Karlsruhe (TH), Germany, oct 2008. (Rank Core A. Taux d'acceptation=36%)

– **A Model-Based Framework for Statically and Dynamically Checking Component Interactions.** Guillaume Waignier, Prawee Sriplakich, Anne-Françoise Le Meur and Laurence Duchien. In *Proceedings of the ACM/IEEE 11th International Conference on Model-Driven Engineering Languages and Systems* (MODELS 2008), pages 371-385, Toulouse, France, oct 2008. (Rank Core A. Taux d'acceptation=21%)

Workshops internationaux

– **A Framework for Bridging the Gap Between Design and Runtime Debugging of Component-Based Applications.** Guillaume Waignier, Prawee Sriplakich, Anne-Françoise Le Meur and Laurence Duchien. In *Proceedings of the 3rd International Workshop on Models@runtime 2008*, Toulouse, France, oct 2008.
– **Enabling Dynamic Co-Evolution of Models and Runtime Applications.** Prawee Sriplakich, Guillaume Waignier and Anne-Françoise Le Meur. In *Proceedings of the 1st IEEE International Workshop on Model-Driven Development of Autonomic Systems* (MDDAS 2008), pages 1116-1121, Turku, Finland, jul 2008.

Posters

– **CALICO : a Component Assembly Interaction Control Framework.** Guillaume Waignier, Anne-Françoise Le Meur and Laurence Duchien. In *Poster au journées nationales du GDR GPL*, Toulouse, France, jan 2009.

Démonstrations

– **A Framework for Agile Development of Component-Based Applications.** Guillaume Waignier, Estéban Duguepéroux, Anne-Françoise Le Meur and Laurence Duchien. In *the 8th BElgian-NEtherlands software eVOLution seminar* (BENEVOL 2009), Louvain, Belgium, dec 2009.

Livrables du projet FAROS

– **Démonstrateur de l'application DMP.** Anne-Françoise Le Meur, Guillaume Waignier, Damien Fournier, Julie Jacques and David Delerue. In *Livrable RNTL FAROS, F-4.6*, 49 pages, sept 2009.
– **Guide pour l'écriture des transformation pivot vers plates-formes.** Noël Plouzeau, Nicolas Ferry, Mireille Blay-Fornarino, Anne-Françoise Le Meur, Sébastien Mosser, Lionel Seinturier, Jean-Yves Tigli and Guillaume Waignier. In *Livrable RNTL FAROS, F-2.5*, 49 pages, jul 2009.
– **Transformations depuis les modèles métier.** Philippe Lahire, Michel Dao, Mireille Blay-Fornarino, Nicolas Rivierre, Philippe Collet, Noël Plouzeau, Sébastien Mosser, Guillaume Waignier, Bruno Traverson and Anne-Françoise Le Meur. In *Livrable RNTL FAROS, F-2.4*, 42 pages, jul 2008.
– **Spécification du méta-modèle pivot**. Noël Plouzeau, Franck Chauvel and Guillaume Waignier. In *Livrable RNTL FAROS, F-2.1*, 41 pages, jul 2008.
– **Métamodèles métiers : production de métamodèles métiers prenant en compte les concepts de contrats**. Philippe Lahire, Guillaume Waignier, Franck Chauvel, Michel Dao, Nicolas Rivierre, Mireille Blay-Fornarino, Bruno Traverson, Noël Plouzeau, Philippe Collet, Sébastien Mosser and Anne-Françoise Le Meur. In *Livrable RNTL FAROS, F-2.2*, 69 pages, dec 2007.
– **Spécification d'une architecture pour la contractualisation de services : expression du besoin**. Anne-Françoise Le Meur, Philippe Lahire, Guillaume Waignier, Philippe Collet, Noël Plouzeau, Michel Dao, Laurence Duchien, Alain Ozanne and Nicolas Rivierre. In *Livrable RNTL Faros, F-1.2.*, 30 pages, jun 2007.

Première partie

État de l’Art

Chapitre 2

Contexte de travail

Sommaire

2.1 Architectures logicielles **11**
2.1.1 Architecture à composants 12
2.1.2 Exemple de modèles à composants 13
2.1.3 Architecture orientée services 16
2.1.4 Architecture hybride SCA 17
2.1.5 Bilan 18
2.2 Développement du logiciel **21**
2.2.1 Modèles de cycles de développement 22
2.2.2 Etape d'analyse des besoins 24
2.2.3 Etape de conception 25
2.2.4 Etape d'implémentation 30
2.2.5 Etape de validation 33
2.2.6 Etape de déploiement 35
2.3 Conclusion **36**

COMME nous l'avons vu dans le chapitre d'introduction, l'objectif de cette thèse est de repenser le moyen d'élaborer une architecture logicielle afin de permettre son évolution fiable. Ce chapitre donne le contexte de notre travail. La section 2.1 présente les deux grandes styles d'architectures logicielles classiquement rencontrés : les architectures à composants et les architectures orientées services. La section 2.2 décrit brièvement les différentes étapes du cycle de développement d'un logiciel. Chacune des étapes du cycle est illustrée sur des exemples d'architectures à composants et d'architectures orientées services.

2.1 Architectures logicielles

La notion d'architecture logicielle est apparue après le paradigme objet [MT00]. Les architectures logicielles reposent sur les principes de *modularisation* et de *réutilisation* du génie logiciel [GJM91]. Grâce à ces deux principes, les architectures logicielles permettent la construction de larges applications en réutilisant des *entités* existantes, qui sont des composants ou des services, et en les *composant* entre elles. Une définition bien connue du terme architecture logicielle est proposée par l'association de standard IEEE :

"*Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.*"

Bien qu'il n'y ait pas de définition universelle, il est usuellement accepté que la spécification d'une architecture logicielle doit décrire au moins les entités contenues dans l'architecture ainsi que les relations entre ces entités.

Dans la suite de ce document, nous nous intéressons aux deux grandes styles d'architectures logicielles qui sont utilisés dans le monde industriel : les architectures à composants [Szy97] et les architectures orientées services [Erl05] qui sont, respectivement, détaillées dans les sections 2.1.1 et 2.1.3. La section 2.1.5 dresse une comparaison entre ces deux styles.

2.1.1 Architecture à composants

Dans les architectures à composants, l'entité composable est le composant. Szypersky a donné une définition, communément admise, de la notion de composant :

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [Szy97].

Cette définition met en avant trois propriétés fondamentales d'un composant logiciel. Premièrement, un composant est une entité qui décrit explicitement les fonctionnalités qu'il offre par le biais d'interfaces contractualisées, ainsi que ses dépendances, c'est-à-dire les fonctionnalités qu'il requiert. Deuxièmement, un composant est une entité composable. Concrètement cela signifie qu'une application à base de composants est en fait un assemblage de composants. De plus, cette composition doit satisfaire les exigences décrites par les interfaces contractualisées des composants. Troisièmement, un composant est une entité capable d'être déployée sur une plate-forme d'exécution, indépendamment des autres composants.

La concrétisation des architectures à composants a donné lieu à la définition de plusieurs modèles à composants, aussi bien dans un contexte académique que dans le monde industriel. Chacun de ces modèles repose sur des concepts communs, à savoir les notions de *composant*, de *port*, de *connecteur* et de *configuration* [MT00]. Cependant, ces concepts peuvent avoir des noms différents et chacun de ces modèles apporte leur lot de règles à suivre pour assembler les composants entre eux :

Composant. Le composant est une entité de calcul ou de stockage. Il peut être vu comme une boîte blanche ou noire, selon que le contenu du composant est explicité ou pas. Il possède des ports, aussi appelés interfaces contractualisées, qui sont les points d'accès du composant, c'est-à-dire que le composant *interagit* avec les autres composants exclusivement au travers de ses ports. Dans la majorité des modèles à composants, les ports peuvent être requis ou fournis, décrivant respectivement les fonctionnalités requises ou fournies par le composant [WLD07a]. D'autres modèles à composants offrent des ports complexes regroupant à la fois des fonctionnalités requises et fournies, comme SafArchie [Bar05] ou UML2 [Obj07b].

Un composant peut être primitif ou composite. Un composant primitif est un composant indivisible. Un composant composite est vu comme une boîte blanche dont le contenu est constitué d'un assemblage de composants.

Connecteur. Un connecteur modélise les interactions entre les composants. Il peut prendre différentes formes suivant le modèle à composants [MB05]. Par exemple il peut être un simple lien qui représente une communication par appel de méthode ou par envoi de message. Il peut aussi modéliser une interaction plus complexe, par exemple, le connecteur peut être chargé d'effectuer une répartition de charge ou établir une communication transactionnelle. Un connecteur connecte un ou plusieurs ports requis avec un ou plusieurs ports fournis *compatibles*. La notion de compatibilité entre les ports est un point crucial. Elle est définie dans les règles de composition du modèle à composants utilisé. Nous illustrons ce point dans la section 2.1.2 en se basant sur trois exemples de modèles à composants.

Configuration. La configuration [MT00] est un terme historique utilisé pour nommer l'assemblage de composants correspondant à l'application. La description de l'assemblage est structurelle et repose sur la création de connecteurs entre composants. Dans les modèles à composants hiérarchiques, c'est-à-dire offrant la notion de composant composite, la configuration est souvent confondue avec le composant composite de plus haut niveau hiérarchique.

Un assemblage de composants est *cohérent* si toutes les règles de composition du modèle à composants sont satisfaites. La cohérence d'une architecture n'inclut donc pas seulement que tous les connecteurs de l'architecture connectent des ports compatibles, mais prend aussi en compte les règles de composition hiérarchique qu'il faut respecter lors de la création des composants composites. Par exemple, dans la majorité des modèles hiérarchiques, un composant doit être contenu dans exactement un composite. Dans le cadre de notre travail, nous souhaitons élaborer un cadre de développement agile et fiable pour les architectures à composants et orientées services. Donc, nous portons un intérêt tout particulier à l'étude des règles de compositions dans différents modèles à composants.

Les architectures à composants peuvent être décrites soit par un langage de description d'architecture (ADL pour "*Architecture Description Language*"), soit graphiquement. Pour des raisons de clarté, nous utilisons la représentation graphique dans la suite du document. Par exemple, la figure 2.1 représente graphiquement un composant composite `HelloWorld` constitué de deux sous composants. Le client `Client` dispose de deux ports, un port fourni `P2` à gauche et un port requis nommé `P3` à droite. Le port requis `P3` est connecté à l'aide d'un connecteur au port fourni `P4` du composant `Server`.

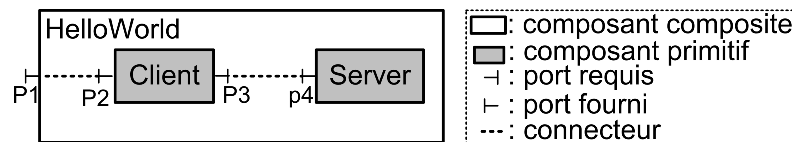

FIG. 2.1: Exemple d'une architecture à composants

2.1.2 Exemple de modèles à composants

Cette section illustre la large variété des règles de composition spécifiques à chacun des modèles à composants à l'aide de deux exemples de modèles à composants académiques (Fractal [BCS04] et OpenCOM [CBG+04]) et d'un exemple de modèle à composants industriels (CCM [Obj02b]). Pour chacun de ces modèles, nous identifions les concepts communs des modèles à composants et nous expliquons leurs règles de composition. Ces trois modèles à composants sont utilisés dans toute la suite de ce document pour illustrer nos propos.

A) Le modèle à composants OpenCOM

OpenCOM est un modèle à composants académique développé à l'Université de Lancaster [CBG+04]. Dans sa première version, OpenCOM repose sur la technologie COM (*Component Object Model*) de Microsoft. Ce modèle est indépendant des langages de programmation et a donné lieu à deux plates-formes d'exécution, une en C++ et une en Java. Ce modèle à composants reste avant tout un prototype de recherche utilisé principalement dans un contexte académique, comme par exemple à l'Université de Lancaster et dans des Universités brésiliennes, mexicaine et néerlandaise. Ce modèle sert aussi de base dans le projet de recherche européen IST RUNES [1] [BTA07] (RUNES pour "*Reconfigurable Ubiquitous Networked Embedded Systems*").

Le modèle à composants OpenCOM, comme tous les modèles à composants, possède les notions de composant, port, connecteur et configuration. Ce modèle n'est pas hiérarchique et ne

[1] http ://www.ist-runes.org

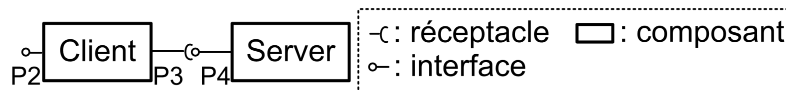

FIG. 2.2: Exemple d'une architecture à composants OpenCOM

supporte donc que les composants primitifs. Les ports fournis et requis sont appelés respectivement interfaces et réceptacles. Dans la projection en Java du modèle, les ports sont des interfaces Java. Les connecteurs correspondent à des connexions entre les interfaces et les réceptacles et ils représentent des invocations par appel de méthodes. La configuration correspond à l'assemblage de composants, comme nous pouvons le voir dans la figure 2.2 qui décrit l'assemblage de composants `Client` - `Server` introduit dans la section 2.1.1.

OpenCOM possède trois règles de composition principales :

- Un réceptacle est *compatible* avec une interface si et seulement s'ils ont le même type.
- Un réceptacle doit obligatoirement être connecté avec une interface compatible, c'est-à-dire qu'il ne doit pas exister de réceptacle non connecté dans une architecture.
- Un réceptacle peut être connecté à plusieurs interfaces s'il est déclaré comme étant à pointeurs multiples (*"multi-pointer"*).

La seconde version du modèle à composants OpenCOM ajoute la prise en charge de la composition hiérarchique en introduisant quatre nouveaux concepts : les *capsules*, les *caplets*, les *binders* et les *loaders*. Une capsule est un conteneur qui contient et gère les composants applicatifs. Une caplet est une partie d'une capsule. Son rôle est similaire à celui d'un composant composite. Les binders et les loaders permettent, respectivement, de connecter les composants entre eux et de charger les composants dans des caplets. Nous ne détaillons pas cette seconde version car elle se rapproche du modèle à composants Fractal qui est décrit ci-dessous.

B) Le modèle à composants Fractal

Fractal est un modèle à composants académique, développé initialement par France Telecom R&D et l'INRIA [BCS04]. Il fait actuellement partie d'un projet actif du consortium international OW2 [2]. Ce modèle à composants a été défini indépendamment d'un langage de programmation, permettant ainsi la projection de ce modèle dans différents langages de programmation, comme C [FSLM02], C# [SPED06] ou Java [BCL+04]. Ce modèle à composants est utilisé, en grande majorité, au niveau national pour concevoir une large variété de logiciels, allant des systèmes d'exploitation [FSLM02], aux applications métier [Dub08] en passant par des plates-formes d'exécution [Abd05, SMF+09] [3]. Dans ce document, nous nous concentrons sur la projection en Java et sa plate-forme d'exécution de référence Julia [BCL+04].

Le modèle à composants Fractal possède les concepts communs de composant, port, connecteur et configuration. Ce modèle supporte à la fois les composants primitifs et composites. Dans ce modèle, les ports fournis et requis sont appelés respectivement, des interfaces serveurs et clients. Dans la projection Java du modèle, ces interfaces sont associées à des interfaces Java. Un connecteur Fractal est un simple lien, nommé *binding*, entre une interface client et une interface serveur. La configuration correspond au composant composite de plus haut niveau hiérarchique. La figure 2.3 est la représentation graphique Fractal du composite `HelloWorld` présenté dans la section précédente.

Les règles de composition du modèle à composants Fractal ont été formalisées quatre ans après la création de la plate-forme d'exécution [MS08]. Nous retenons quatre règles principales :

- Une interface client est *compatible* avec une interface serveur si et seulement si le type de l'interface serveur est de même type ou un sous type de l'interface client.
- Une interface client déclarée obligatoire (*"mandatory"*) doit impérativement être connectée avec une interface serveur compatible.

[2] http ://fractal.ow2.org
[3] http ://fractal.ow2.org/users.html

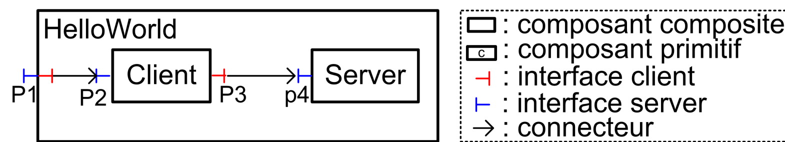

FIG. 2.3: Exemple d'une architecture à composants Fractal

– Une interface client peut être connectée à plusieurs interfaces serveur uniquement si l'interface client est déclarée *collection*.
– Un même composant peut être contenu dans différents composants composites grâce au support de composant partagé de Fractal. Ce support de partage de composant permet de modéliser le partage de ressources ou de services. A notre connaissance, Fractal est le seul modèle à composants possédant ce concept.

C) Le modèle à composants CORBA

Le modèle à composants CORBA (CCM pour *"CORBA Component Model"*) est un modèle à composants industriel défini par le consortium international OMG (*"Object Management Group"*) [Obj02b]. C'est une extension du modèle objet CORBA (*"Common Object Request Broker Architecture"*) [Obj02a]. Ce modèle à composants a la particularité de supporter plusieurs langage de programmation et d'être interopérable, permettant ainsi la communication entre des composants CORBA implémentés dans des langages différents, comme Java, C ou C++. Dans la suite du document, nous nous focalisons sur la plate-forme d'exécution OpenCCM qui est une implémentation Java de la version 3 du modèle et qui est hébergée par le consortium OW2 [4] [BCM04].

La version 3 du modèle à composants CORBA repose aussi sur les concepts de composant, port, connecteur et configuration (cf. figure 2.4). Cependant, elle ne prend pas en charge la hiérarchisation des composants et ne supporte donc que les composants primitifs. En outre, ce modèle fait une distinction entre les ports permettant une communication synchrone par appel de méthode et ceux permettant une communication asynchrone par envoi de messages. Les ports fournis et requis synchrones sont appelés respectivement facette et réceptacle, les ports fournis et requis asynchrones sont appelés source et puits d'événements. Le connecteur CCM correspond à une simple liaison et la configuration correspond au descripteur d'assemblage des composants.

La version 4 du modèle introduit le concept de composition [Obj06a]. Cependant, ce concept est différent du concept de composant composite dans la mesure où une composition n'est pas considérée comme un composant. En conséquence, ce modèle ne permet pas de créer des assemblages de composants de manière récursive.

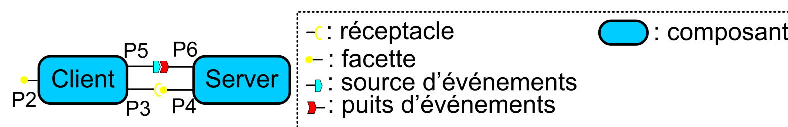

FIG. 2.4: Exemple d'une architecture à composants CORBA

Les règles de composition de CCM sont plus complexes que celles des deux modèles précédents, notamment en raison de son support explicite des communications synchrones et asynchrones. Nous citons trois exemples de règles de composition :
– Un réceptacle doit obligatoirement être connecté à une facette de même type.
– Un puits d'événements peut être connecté à une ou plusieurs sources d'événements de même type.

[4] http ://openccm.ow2.org

– Un réceptacle peut être connecté à plusieurs facettes si ce réceptacle est déclaré comme étant multiple.

D) Bilan

De nombreux modèles à composants différents coexistent. Néanmoins tous ces modèles reposent sur les mêmes concepts : les concepts de composant, de port, de connecteur et de configuration. De plus, dans tous les modèles, une architecture à composants correspond à une spécification structurelle qui décrit l'assemblage de composants et de connecteurs. Les différences entre les modèles à composants résident dans le mode de communication entre les composants et dans les règles de composition. Concrètement, chaque modèle offre son propre mode de communication, qui peut-être par exemple synchrone par appel de méthodes ou asynchrone par envoi de messages. De même, chaque modèle définit ses propres règles de compatibilité entre les composants et ses règles de composition hiérarchique. Cependant, en dépit de ces différences entre les modèles, la manière de raisonner pour concevoir une architecture est la même : c'est un raisonnement structurel.

2.1.3 Architecture orientée services

L'autre grand style d'architectures logicielles correspond aux architectures orientées services. Il existe de nombreuses définitions du terme architecture orientée services (SOA pour *"Service Oriented Architecture"*) provenant de différents organismes, comme le consortium OASIS [OAS08] ou le consortium W3C (*"World Wide Web Consortium"*). Par exemple, la définition donnée par le W3C est :

"A service Oriented Architecture is a set of components which can be invoked and whose interface descriptions can be published and discovered" [5].

Cette définition exprime qu'une architecture SOA est la composition d'un ensemble de composants. Ces composants, qui sont par convention appelés des services, exportent les fonctionnalités qu'elles offrent à travers des interfaces. Dans la suite du document, nous nous focalisons sur les services WEB qui sont une mise en œuvre normalisée du paradigme SOA dédié au WEB [Erl05] :

Service. Un service est une action effectuée par un fournisseur de service, comme une entreprise, et produit un résultat final à un consommateur de service [Erl05]. Le service possède donc un point d'accès fourni. Dans le cas des services WEB, ce point d'accès et appelé un port et doit être spécifié avec le langage standardisé WSDL (*"Web Service Definition Language"*) [W3C01]. Il permet de décrire le service, les opérations qu'il offre et les messages échangés.

Le service diffère de la notion de composant. Notamment, un service n'explicite pas ses dépendances, seules les fonctionnalités offertes sont décrites. De plus, le concept de hiérarchisation n'existe pas, c'est-à-dire qu'un service est toujours "primitif".

Communication. Les services sont faiblement couplés entre eux et communiquent uniquement par échange de messages. La connexion entre les services est dynamique car elle est créée juste avant la communication entre les services et est supprimée à la fin de la communication. Dans le cas des Services WEB, le protocole de communication est SOAP (*"Simple Object Access Protocol"*) [W3C07]. C'est un protocole standardisé reposant sur XML [W3C04] qui décrit les messages échangés entre les services de manière indépendante à tout langage de programmation. Ce protocole favorise ainsi l'interopérabilité entre différents services.

[5]http ://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/

Composition. La définition de la composition de service repose sur la spécification des interactions entre les services. La spécification de la composition se focalise sur le comportement des services, c'est-à-dire qu'elle exprime l'ordre d'enchaînement des invocations des autres services avec les messages envoyés et reçus. Aucun formalisme n'a été imposé pour spécifier la composition. L'architecte peut donc utiliser un langage de programmation, comme Java, ou un langage de composition dédié, comme WS-BPEL (*"Web Services Business Process Execution Language"*) [OAS07], qui est présenté dans la suite du document.

Contrairement aux architectures à composants, la composition de service WEB repose sur l'utilisation de services déjà déployés, c'est-à-dire que ces services sont gérés par des organismes tiers et sont accessibles via le WEB. Afin de faciliter l'accès et la recherche à ces services, des services d'annuaires permettent de rechercher dynamiquement le service désiré.

Bilan

Les architectures orientées services reposent sur les concepts de service, de communication et de composition. La composition correspond à une spécification comportementale qui décrit le flot de contrôle, c'est-à-dire l'enchaînement des invocations des autres services. La construction d'une telle architecture nécessite donc de raisonner sur le comportement du logiciel.

2.1.4 Architecture hybride SCA

Les modèles à composants et orientés services coexistent. Afin de les réunir et de les faire interopérer, le consortium OpenSOA [6], qui est composé de 17 partenaires industriels dont IBM, Sun, Oracle et RedHat, a défini le modèle industriel SCA (*"Service Component Architecture"*) [BII+07]. La spécification finale 1.0 a été publiée en mars 2007. Ce modèle a pour objectif de pouvoir englober un large éventail de technologies. Il se concentre donc sur l'intégration de technologies hétérogènes et leur interopérabilité. Par exemple, cela inclut le support des composants implémentés dans différents langages de programmation, comme Java et C, et différentes technologies, comme les composants OSGI [OSG07], Spring [JHA+08] et JBI [SUN05]. De plus, différents modes de communication entre composants sont aussi pris en compte, comme Java RMI [WRW96], WebService [W3C07] et JMS [SUN03]. Le modèle SCA est adapté au développement de logiciels qui mettent en relation les entreprises et/ou les particuliers car le support multi-protocoles de communication et multi-implémentations facilite l'interopérabilité. Dans la suite du document, nous utilisons la plate-forme d'exécution FraSCAti qui, elle aussi, est développée au sein du consortium OW2 [7] [SMF+09]. Cette plate-forme d'exécution est implémentée à l'aide du modèle à composants Fractal.

Les plates-formes SCA ont aussi la particularité d'être des plates-formes d'exécution de services WEB. En effet, de part son objectif d'être multi technologies, le modèle SCA englobe aussi les technologies propres aux services WEB, comme par exemple le mode de communication des services WEB. En conséquence, les plates-formes SCA supportent l'hébergement de services WEB.

Nous retrouvons dans le modèle SCA les quatre concepts communs des modèles à composants : le concept de composant, de port, de connecteur et de configuration (cf. figure 2.5). Le modèle à composants SCA prend en charge les composants primitifs et composites. Les ports fournis et requis sont appelés respectivement services et références. Les connecteurs sont des liens appelés *wire* et la configuration correspond au modèle d'assemblage SCA (*"SCA assembly model"*).

Les règles de composition du modèle SCA sont complexes. Notamment un connecteur peut connecter un port fourni avec un port requis si et seulement si :

- les opérations du port requis sont de même type ou un supertype des opérations du port fourni.

[6] http ://www.osoa.org
[7] http ://frascati.ow2.org

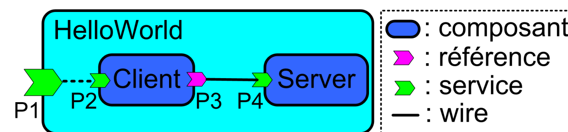

FIG. 2.5: Exemple d'une architecture SCA

- deux opérations sont compatibles si leurs signatures sont compatibles, c'est-à-dire, si elles ont le même nom et les mêmes types de paramètres d'entrées et de sorties.
- l'ordre des paramètres d'entrées et de sorties doivent aussi être identiques
- l'ensemble des exceptions attendues par le port fourni doit être de même type ou un super-type de ceux du port requis.

De plus, en raison du support multi-langages des services et des références, la spécification de la compatibilité entre ports repose sur l'équivalence des types décrits dans les différents langages : "*A Wire can connect between different interface languages (eg. Java interfaces and WSDL portTypes) in either direction, as long as the operations defined by the two interface types are equivalent. They are equivalent if the operation(s), parameter(s), return value(s) and faults/exceptions map to each other*".

Bilan

Le modèle SCA est un modèle hybride qui a pour objectif de réunir les composants et les services WEB. Néanmoins, dans sa façon de raisonner pour construire une architecture, ce modèle se rapproche plus des modèles à composants que des modèles orientés services. En effet, pour concevoir une architecture SCA, il est nécessaire de raisonner sur les concepts de composants, de ports, de connecteurs et de configuration, qui sont des concepts spécifiques aux modèles à composants. De plus, la configuration repose sur une description structurelle de l'assemblage de composants, ce qui est similaire aux modèles à composants. Par ailleurs, le modèle SCA n'offre pas de vue comportementale de l'architecture, comme le fait un modèle orienté services.

Cependant, les plates-formes d'exécution SCA permettent l'hébergement de services WEB. En conséquence, de notre point de vue, SCA est un modèle à composants qui permet de concevoir des services avec une approche composant, c'est-à-dire qu'un service correspond à un assemblage structurel de composants.

2.1.5 Bilan

De nombreux travaux, comme Acme [GMW00], xADL [DdHT01] et SCL [Fab07], ont comparé les différents modèles à composants afin de créer des nouveaux modèles à composants plus généraliste. D'autre travaux se sont focalisés sur la comparaison des architectures à composants et orientées services, comme par exemple [Pet06, BL07]. Les architectures à composants et les architectures orientées services reposent sur les mêmes concepts principaux mais diffèrent sur certains points qui sont résumés dans le tableau 2.1. Ces architectures permettent la conception de larges applications en réutilisant des entités logicielles (composants ou services). Tous les modèles possèdent en commun les quatre concepts suivants : (1) entité, (2) port, (3) communication et (4) composition :

(1) Dans les architectures à composants, une entité est appelée composant. Le composant peut avoir une granularité très fine jusqu'à très grosse, c'est-à-dire qu'il peut aussi bien encapsuler une fonctionnalité élémentaire précise ou une logique métier complexe (cf. colonne 1 du tableau 2.1). Dans les architectures orientées services, une entité est appelée service. Généralement, le service a une grosse granularité, c'est-à-dire qu'il encapsule et expose une fonctionnalité métier qui peut être réutilisée aussi bien dans le cadre de l'entreprise ou entre des entreprises différentes.

(2) Dans les architectures à composants, les composants explicitent leurs dépendances aux travers de ports requis et fournis. Dans le cadre des architectures orientées services, seuls les ports fournis sont rendus explicites, via le fichier de description WSDL. Les ports requis sont

Bilan	(1) Granularité	(2) Couplage	(3) Communication	(4) Composition	(5) Hiérarchique	(6) Particularité	(7) domaine d'application
AADL	fin/gros	fort	flux typée	structurel	oui	couple logiciel/matériel	Système embarqué temps réel
ACME	fin/gros	fort	non défini	structurel	oui	générique	application métier
ArchJava	fin/gros	fort	méthode Java	structurel	oui	couplé avec Java	application métier
C2	fin/gros	fort	message asynchrone	structurel	oui	architecture en couche	interface graphique
CCM	fin/gros	fort	méthode synchrone message asynchrone	structurel	non	multi-langage	application métier
Darwin	fin/gros	fort	flux typé	structurel	oui		application métier
Fractal	fin/gros	fort	méthode synchrone	structurel	oui	partage	application métier plate-forme d'exécution système d'exploitation
Olan	fin/gros	fort	opération synchrone opération asynchrone	structurel	oui		application métier
OpenCOM	fin/gros	fort	méthode synchrone	structurel	non	indépendant du langage	application métier
Rapide	fin/gros	fort	événement synchrone événement asynchrone	comportemental	oui	simulation	application métier
SafArchie	fin/gros	fort	méthode synchrone	structurel	oui	vérification	application métier
SCA	fin/gros	fort/faible	multiple	structurel	oui	multi-technologie	système d'information
Service Web	gros	faible	message asynchrone	comportemental	non	multi-langage	système d'information
SOFA	fin/gros	fort	méthode synchrone méthode asynchrone	structurel	oui	protocole comportemental	application métier
UniCon	fin/gros	fort	méthode/message/flux synchrone et asynchrone	structurel	oui		application métier
Wright	fin/gros	fort	messages synchrone message asynchrone	structurel comportemental	oui	CSP	analyse comportemental
xADL	fin/gros	fort	non défini	structurel	oui	générique	application métier

TAB. 2.1: Comparaison des modèles à composants et des modèles à services

masqués dans la description de la composition des services. Par exemple, les ports requis sont référencés sous le nom de *"partnerLink"* dans le cas d'une composition utilisant la description BPEL.

(3) Les architectures à composants et orientées services possèdent en commun le concept de communication qui décrit les interactions entre les composants et les services (cf. colonnes 2 et 3). Néanmoins, selon le modèle à composants ou orienté services, le mode de communication diffère. Dans les architectures à composants, une communication entre composants est modélisée par un connecteur qui représente un lien structurel entre un port requis et un port fourni. Les composants sont donc fortement couplés entre eux. Les composants supportent l'échange de messages, c'est-à-dire de données, ainsi que l'échange d'objet, c'est-à-dire de données et d'opérations de traitement. Le mode de communication peut correspondre à des appels de méthode synchrones, comme dans OpenCOM et Fractal, ou à des envois de messages asynchrones. Certains modèles à composants supportent différents modes de communication comme OpenCCM qui gère, à la fois, les communications synchrones par appels de méthodes et les communications asynchrones par envois de messages. Dans les architectures orientées services, les services sont faiblement couplés entre eux. Les services ne supportent que le mode de communication asynchrone par envoi de messages. Quant au modèle hybride SCA, il gère potentiellement tous les modes de communication, en fonction des fonctionnalités implémentées dans la plate-forme d'exécution utilisée. Concrètement, il prend en charge le couplage fort des composants et le couplage faible des services. Par ailleurs, en raison de ces différences de communication, chaque modèle à composants et orienté services possède ses propres règles de compatibilité entre les ports requis et fournis des composants et des services. Par exemple, ces règles de compatibilité peuvent reposer sur la correspondance exacte des types des ports requis et fournis, comme dans OpenCOM, ou elles peuvent être plus souples et se baser sur une relation de sous typage entre les types de ports, comme dans Fractal.

(4) Une architecture logicielle à composants et orientée services résulte d'une composition des entités (composants ou services). Dans le cas des architectures à composants, la composition est structurelle, c'est-à-dire que l'architecture correspond à un assemblage de composants (cf. colonne 4). De plus, certains modèles gèrent la composition hiérarchique, comme le modèle Fractal ou SCA, alors que les autres modèles sont plats, comme OpenCOM, OpenCCM (cf. colonne 5). Mais aussi, chaque modèle dispose de quelques atouts spécifiques (cf. colonne 6). Par exemple, une particularité du modèle Fractal est la gestion du partage des composants, et celle de SCA est de permettre l'utilisation d'une multitude de technologies différentes. Dans le cas des architectures orientées services, la composition est comportementale et ne prend pas en compte la hiérarchisation. Donc, contrairement aux architectures à composants, la spécification de la structure d'une architecture orientée services n'est pas la préoccupation principale. La structure peut-être déduite à partir de la spécification du comportement, c'est-à-dire en identifiant toutes les communications potentielles entre les services.

En conséquence, chacun de ces modèles a ses spécificités propres, ce qui rend son utilisation pertinente pour le développement de certaines applications et inapte pour d'autres (cf. colonne 7). Par exemple, les plates-formes à composants CORBA sont utilisables pour l'exécution d'une application dont les composants sont écrits dans différents langages de programmation, ce que la plate-forme d'exécution Fractal Julia ne permet pas de faire. D'un autre coté, la plate-forme Fractal autorise la modélisation hiérarchique de l'application, ce qui peut s'avérer utile pour la structuration du logiciel. Le modèle Fractal a aussi prouvé son efficacité pour le développement de plates-formes d'exécution et de systèmes d'exploitation. Dans le contexte de la thèse, nous souhaitons donc que notre cadre de développement supporte un maximum de modèles et de plates-formes à composants et orientés services différents.

2.2 Développement du logiciel

La section précédente a présenté les architectures à composants et les architectures orientées services. Cette section donne un aperçu du cycle de développement d'une application à base de ces architectures.

Le développement d'un logiciel est effectué par une succession d'étapes et de tâches entreprises par des acteurs différents. Le standard ISO 12207 définit toutes les tâches nécessaires pour le développement logiciel [ISO95]. Ce standard inclut la description des processus techniques, comme le développement du système et sa maintenance, des processus de support, comme la rédaction de la documentation, et des processus organisationnels.

Notre travail se focalise sur le processus technique de développement du système. Le standard ISO 12207 établit un lien fort entre le système et le logiciel. Le système regroupe la totalité des infrastructures concernées par le développement, c'est-à-dire le support matériel, les ressources et les autres logiciels existants qui interagissent avec le logiciel en cours de développement. Le logiciel correspond à l'application métier. Avec le standard ISO 12207, le processus technique de développement du logiciel est composé des étapes suivantes : analyse des besoins du système, analyse des besoins du logiciel, conception du système, conception de l'architecture du logiciel, conception détaillée du logiciel, codage, test du code, test du logiciel complet, test du système et installation du logiciel. Ce standard offre un découpage des étapes de très fine granularité, mais globalement l'élaboration du logiciel inclut les cinq grandes étapes suivantes : l'analyse des besoins, la conception, l'implémentation, la validation et le déploiement. Dans notre approche, nous raffinons l'étape de validation dans la mesure où nos travaux se focalisent sur l'évolution fiable de logiciel. Nous proposons donc de distinguer l'étape de validation statique et l'étape de validation dynamique. Notre approche repose donc sur les étapes du cycle de développement suivants :

- Durant l'**étape d'analyse des besoins**, un analyste logiciel identifie les besoins du client, c'est-à-dire qu'il établit un cahier des charges fonctionnel et technique qui décrit les fonctionnalités et les contraintes du logiciel et du matériel. Cette étape inclut par exemple l'écriture de scénarios d'utilisation du système.
- L'**étape de conception** consiste à décrire le logiciel à un haut niveau d'abstraction, c'est-à-dire sans se soucier des détails d'implémentation. Elle est effectuée par un architecte logiciel.
- Pendant l'**étape d'implémentation**, les développeurs écrivent le code du logiciel.
- Nous proposons de raffiner l'**étape de validation** en deux étapes distinctes : la validation statique et la validation dynamique.
  - L'**étape de validation statique** consiste à vérifier que la conception de l'architecture est cohérente, c'est-à-dire qu'elle respecte les règles de composition du modèle.
  - Lors de l'**étape de validation dynamique**, les testeurs exécutent le logiciel afin de s'assurer que le comportement du logiciel et du matériel est conforme à ce qui a été défini lors de l'étape d'analyse des besoins. Le standard ISO ne définit aucun moyen de validation. Par exemple, les testeurs peuvent exécuter les scénarios d'utilisation établis durant l'étape d'analyse des besoins. Cette étape nécessite que le logiciel soit déployé.
- L'**étape de déploiement** correspond à l'installation du logiciel sur les différentes machines cibles par un administrateur système.

Le standard ISO définit aussi le processus technique de maintenance du logiciel. Ce processus est activé lorsque le code du système logiciel doit être modifié suite à la découverte d'une erreur ou d'un besoin d'évolution du logiciel. L'objectif est de modifier le système tout en gardant son intégrité. Lorsque le logiciel doit évoluer, le processus technique de développement du système, décrit précédemment, est réexécuté.

Le standard ISO 12207 ne spécifie pas d'ordre temporel entre les étapes de développement. Au contraire, il permet la définition de différents enchaînements possibles de ces étapes, appelés des modèles de cycle de développement du logiciel. Par exemple, l'exécution de ces étapes peut être séquentielle, itérative ou se chevaucher. Pour illustrer l'étendue des différents modèles de cycles de développement, nous présentons trois exemples de modèles de cycle dans la section 2.2.1.

2.2.1 Modèles de cycles de développement

Il existe de nombreux modèles de cycles de développement différents. Les premiers modèles de cycles de développement existants sont très linéaires et non flexibles, c'est-à-dire qu'une fois l'étape complètement finie, l'étape suivante est enchaînée. Les modèles de cycles de développement suivants ont évolué afin d'être de plus en plus réactifs et de permettre l'évolution du logiciel. Dans la suite de la section, nous présentons deux exemples de modèle de cycle linéaire, puis nous finissons avec des exemples de modèles de cycle plus réactifs qui facilitent l'évolution du logiciel.

A) Cycle descendant

Cette section présente deux cycles de développement descendant : le cycle de développement en cascade et le cycle de développement en V. Elle finit par une discussion des avantages et des inconvénients de ces cycles de développement.

Cycle de développement en cascade. Le cycle de développement en cascade, appelé modèle *waterfall*, a été introduit par Royce en 1970 [Roy70]. Ce cycle, représenté à la figure 2.6, est constitué de six étapes successives : La première étape correspond à l'analyse des besoins matériels, suivie par l'analyse des besoins du logiciel. L'étape de conception, puis l'étape de codage arrivent ensuite. L'étape de codage correspond à l'étape d'implémentation dans le standard ISO. Ensuite l'étape de test, correspondant à l'étape de validation dynamique, est exécutée. Le cycle se termine par l'étape de déploiement, qui est appelée opération.

Ce cycle de développement suggère d'effectuer les étapes de développement successivement. L'étape suivante est exécutée après avoir complètement finie l'étape précédente. Par exemple, l'étape d'implémentation n'est effectuée qu'après avoir entièrement conçu le système et s'être assuré que la conception est correcte. Lorsqu'un problème est détecté dans une étape, il est permis de revenir sur l'étape directement précédente. Une version améliorée du modèle permet aussi de revenir sur des étapes non consécutives.

D'autre part, ce cycle de développement est dirigé par la documentation, c'est-à-dire qu'à chaque étape de développement une spécification complète doit être écrite. Cette spécification sert de base pour l'étape suivante.

Cycle de développement en V. Le cycle de développement en V a été développé par la République Fédérale d'Allemagne et est utilisé depuis les années 1980 [IAB97]. Ce cycle peut être vu comme une extension du cycle de développement en cascade. Néanmoins, au lieu de développer séquentiellement le logiciel, puis les tests, comme cela est fait dans le modèle de

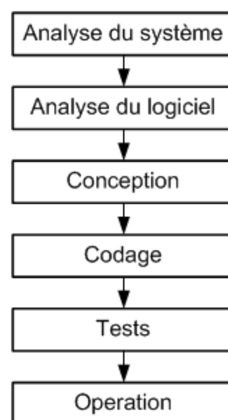

FIG. 2.6: Cycle en cascade

cycle en cascade, dans le modèle en V, les tests sont écrits en parallèle (cf. figure 2.7). Ce cycle associe donc à chaque étape de conception, une étape de validation. L'objectif du cycle est de limiter, en cas de détection d'anomalies, le retour sur des étapes précédentes.

Le cycle en V est composé de neuf étapes. L'analyste logiciel identifie les besoins du logiciel pendant l'étape d'analyse des besoins. En même temps, l'analyste décrit les tests de recette qui permettront de valider le logiciel par l'approbation du client lors de l'étape de la recette. Durant cette étape de validation, le client teste le logiciel pour identifier si le logiciel répond bien à tous les besoins qu'il a exprimés durant l'étape d'analyse des besoins. L'étape de conception du système est validée avec l'étape de test du système. Ce test correspond à identifier si le logiciel conçu s'intègre correctement au sein de l'infrastructure informatique du client, par exemple si le logiciel interagit bien avec les autres logiciels déjà existants. Durant l'étape de conception de l'architecture, l'architecte logiciel décrit le logiciel en termes de composition d'entités (composants ou services). L'architecte définit aussi les tests qui permettent de valider la composition des ces entités lors de l'étape des tests d'intégration. L'étape de conception détaillée du logiciel correspond à l'expression de la spécification des entités. Elle est associée à l'étape de test unitaire vérifiant la conformité de l'implantation des entités avec leur spécification. Enfin l'étape de codage est similaire à l'étape d'implémentation.

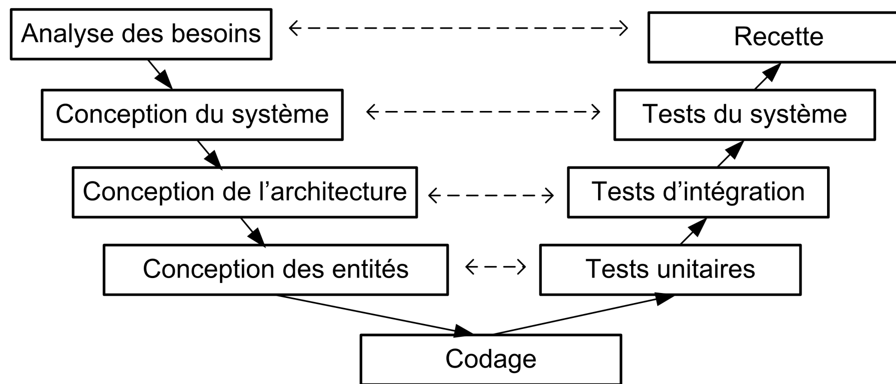

FIG. 2.7: Cycle en V

Discussion. Les cycles de développement précédents sont linéaires. Ils possèdent un inconvénient majeur. En effet, ils partent de l'hypothèse de pouvoir effectuer une étape uniquement en se basant sur les étapes précédentes. Notamment, ils suggèrent une identification complète des besoins avant de commencer la conception. Or, il n'est pas évident pour un client de pouvoir exprimer parfaitement ses besoins. Généralement, le client a besoin de voir un prototype fonctionnel pour commenter les fonctionnalités qu'il désire. Ces modèles de cycles sont donc rigides et ne sont pas adaptés pour développer des logiciels qui évoluent rapidement.

B) Cycle de développement itératif et incrémental

Afin d'introduire plus de souplesse dans le cycle de développement, les modèles de cycle de développement ont évolué et sont devenus itératifs et incrémentals [LB03]. Ces modèles de cycle évitent d'être linéaires, séquentiels et dirigés uniquement par la documentation. Globalement, le logiciel est développé en effectuant plusieurs itérations apportant chacune un incrément. Chaque itération correspond à un enchaînement des étapes de développement. Généralement, l'enchaînement inclut les cinq grandes étapes, à savoir l'analyse des besoins, la conception, l'implémentation, le déploiement et la validation (voir la figure 2.8). Un incrément est un raffinement du logiciel, créé dans l'itération précédente, par ajout de nouvelles fonctionnalités. L'idée générale vise à diminuer la complexité du développement du logiciel en créant d'abord un logiciel avec moins de fonctionnalités et de s'assurer que les fonctionnalités implémentées sont

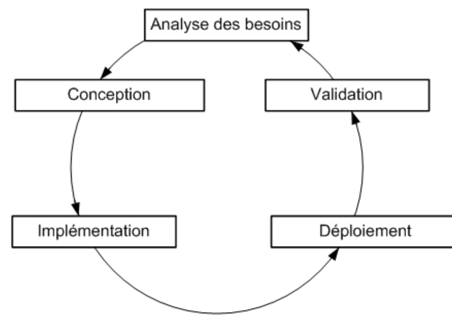

FIG. 2.8: Cycle itératif et incrémental

correctes, en effectuant des tests, avant d'ajouter les autres fonctionnalités au fur et à mesure dans les itérations suivantes.

Il existe de nombreux modèles de cycle itératif et incrémental [AWSR03]. Suivant les modèles, le temps de développement consacré à effectuer une itération est variable et peut aller de quelques heures à quelques mois. Les premiers modèles de cycles de développement itératifs et incrémentals ont été utilisés de manière empirique depuis 1960 pour la conception de gros systèmes par des sociétés comme IBM ou la NASA [LB03]. Par exemple, le projet Mercury de la NASA utilisait des itérations de durée équivalente à une demi-journée. Ce cycle de développement était dirigé par les tests, c'est-à-dire que les tests étaient écrits avant chaque micro-incrément. Ce n'est qu'aux environs de l'an 2000, que le cycle de développement itératif et incrémental acquiert une visibilité et devient populaire. Il est notamment incorporé dans les nouvelles méthodologies de développement, comme l'*Extrem Programming* (XP) en 1996 [Bec99], la méthode de développement de systèmes dynamiques (DSDM) en 1997 [Sta97] et SCRUM en 1999 [BDS+99].

Finalement en 2001, 17 experts en cycle de développement, auteurs des méthodologies DSDM, XP, SCRUM, etc., se sont réunis pour promouvoir le cycle de développement itératif et incrémental. Lors de cette réunion, ils ont rédigé le manifeste agile [8] et défini le terme *"développement agile"*, pour nommer un développement basé sur un cycle itératif et incrémental. Le manifeste agile prône quatre valeurs fondamentales regroupées au sein de 17 principes. Ces valeurs suggèrent de mettre l'accent sur le développement d'un logiciel fonctionnel plutôt que d'écrire une documentation complète. De plus, le client doit être directement impliqué dans le processus de développement et doit fournir un retour continu sur le logiciel en fonction de ses attentes. En accord avec le retour du client, le logiciel doit évoluer pour mieux satisfaire les besoins du client. En outre, le logiciel a besoin d'être validé à chaque étape du cycle de développement.

Après avoir donné un aperçu de quelques modèles de cycle de développement, nous détaillons les différentes étapes d'un cycle de développement dans les sections 2.2.2 à 2.2.6, c'est-à-dire les étapes d'analyse des besoins, de conception, d'implémentation, de validation et de déploiement.

2.2.2 Etape d'analyse des besoins

Durant l'étape d'analyse des besoins, les analystes définissent le cahier des charges fonctionnel et technique. Ils existent différentes normes qui proposent des méthodes de rédaction du cahier des charges, comme la norme AFNOR X 50-151 [AFN07]. L'objectif de cette étape vise à déterminer la viabilité et la faisabilité du projet, puis de lister les fonctions du logiciel et ses contraintes. Différents outils facilitent le travail de l'analyste, comme ceux définis par la méthode APTE® [apt00]. Par exemple, le diagramme de la *"bête à corne"* permet de modéliser le besoin et

[8] http ://agilemanifesto.org

le diagramme *"pieuvre"* permet de définir les fonctions du système, ainsi que les relations entre ces fonctions. Dans le cadre de ce document, nous ne traitons pas l'étape d'analyse des besoins.

2.2.3 Etape de conception

L'étape de conception d'un logiciel succède directement à l'étape d'analyse des besoins. Durant cette étape, l'architecte logiciel a besoin de décrire globalement l'architecture logicielle sans se soucier des détails d'implémentation, c'est-à-dire que l'architecte spécifie la composition des entités. Or, selon le style d'architectures logicielles, c'est-à-dire à composants ou orientées services, la manière de raisonner pour concevoir l'architecture est différente, comme nous l'avons présenté dans la section 2.1.5.

A) Méthode de raisonnement

Architecture à composants. Dans le cas des architectures à composants, la conception de l'architecture consiste à effectuer un raisonnement structurel pour assembler les composants entre eux. L'architecte a donc besoin de manipuler les concepts de composants, de ports et de connecteurs. Concrètement, il définit les composants avec leurs ports, puis il connecte les ports des composants grâce à des connecteurs. Durant cette étape, il peut aussi réutiliser des composants existants. Dans un souci de faciliter la recherche des composants adéquats, plusieurs travaux proposent de sélectionner des composants en fonction des caractéristiques désirées [GFS08, VNdARdS09]. De plus, l'architecte a aussi besoin de s'assurer que la structure de son architecture est correcte, c'est-à-dire qu'elle respecte les règles de composition du modèle à composants utilisé.

Architecture orientée services. Dans le cas des architectures orientées services, la conception correspond à décrire le flot de contrôle, c'est-à-dire l'enchaînement des invocations des services. Concrètement l'architecte spécifie des nouveaux services, recherche des services existants et décrit le flot de contrôle de l'architecture. L'architecte a donc besoin de pouvoir raisonner sur les concepts de services et sur le comportement de l'architecture.

Aide à la conception. Pour faciliter la conception d'architectures logicielles, deux grandes approches qui reposent sur le grand principe d'*abstraction* du génie logiciel, sont actuellement très utilisées : les langages dédiés et l'ingénierie dirigée par les modèles. Ces approches visent à abstraire les concepts requis pour la conception du logiciel afin de faciliter le raisonnement et la construction de l'architecture. Toutes les informations non pertinentes pour le raisonnement sont alors masquées. Toute la difficulté réside dans la sélection des informations pertinentes. De plus, ces approches permettent aussi de vérifier que l'architecture décrite respecte les règles de composition du modèle utilisé. La suite de cette section détaille ces deux approches.

B) Langages dédiés

Les langages dédiés (DSL pour *"Domain Specific Language"*) sont, par définition, des langages spécifiques à un domaine [vKV00]. Un DSL est une notation de haut niveau, efficace et complète permettant la description d'un domaine d'application. Les DSLs sont utilisés dans de nombreux domaines différents, comme les systèmes d'exploitation [PBC+97] ou le domaine vidéo [TMC97].

Les DSLs sont aussi très répandus dans le domaine de la conception d'architectures logicielles, et plus particulièrement pour la conception d'architectures à composants ou orientées services. Dans ce domaine, les DSLs sont appelés des langages de description d'architecture (ADL) [MT00]. Ils possèdent deux avantages principaux : la simplification de la description de l'architecture et la vérification de cette description. En effet, l'ADL reflète directement les concepts requis pour la conception d'architecture logicielle permettant ainsi une meilleure prise en main

par un architecte. De plus, de par sa syntaxe dédiée, l'ADL permet une description plus concise de l'architecture et offre des abstractions plus appropriées que ne le ferait un langage générique.

D'autre part, presque chaque modèle offre à l'architecte un ADL dédié. Ainsi, les outils d'analyse du langage sont spécifiques à chaque modèle et peuvent prendre en compte les règles de composition du modèle. Les ADLs permettent donc une validation à la fois syntaxique et sémantique de la description de l'architecture.

Architecture à composants. Dans le cas des architectures à composants, la description de haut niveau correspond à spécifier la structure de l'architecture. Afin de permettre cette description, les ADLs explicitent clairement les concepts structurels, à savoir les concepts de composant, port, connecteur et configuration. De plus, la majorité des ADLs sont capables de vérifier que les règles de composition sont respectées. Notamment, ils vérifient statiquement la cohérence de l'assemblage et la compatibilité des ports connectés entre eux. Si une erreur est détectée, ils produisent un message d'erreur directement compréhensible par l'architecte logiciel et font référence à la règle de composition violée, et cela afin que l'architecte puisse corriger le problème.

Il existe de nombreux modèles à composants différents et la plupart des modèles possèdent chacun leur propre ADL, comme par exemple Fractal ADL pour le modèle à composants Fractal et l'IDL ("*Interface Definition Language*") pour le modèle à composants CORBA. Chacun de ces ADLs utilisent les dénominations propres au modèle à composants. Par exemple, les listings 2.1 à 2.3 sont des extraits de la spécification de l'assemblage des deux composants `Client` et `Server` de la section 2.1.1, à l'aide respectivement de l'ADL Fractal, OpenCCM et SCA. Comme le montrent les listings, chaque ADL utilise un ensemble de mot-clés spécifiques à chaque modèle à composants pour nommer les concepts de composant, port et connecteur. Par exemple, pour définir le port fourni du composant `Client`, l'ADL Fractal utilise le terme d'`interface server` (cf. ligne 6 de la figure 2.1), OpenCCM utilise le terme de `provide` (cf. ligne 3 de la figure 2.2) et SCA utilise le terme `service` (cf. ligne 6 de la figure 2.3).

```
1<?xml version="1.0" encoding="ISO-8859-1" ?>
2<!DOCTYPE definition PUBLIC
3    "-//objectweb.org//DTD Fractal ADL 2.0//EN"
4    "classpath://org/objectweb/fractal/adl/xml/basic.dtd">
5  <definition name="HelloWorld">
6    <interface name="P1" role="server" signature="java.lang.Runnable"/>
7    <interface name="P6" role="client" signature="Service"/>
8    <component name="Client">
9      <interface name="P2" role="server" signature="java.lang.Runnable"/>
10     <interface name="P3" role="client" signature="Service"/>
11     <content class="ClientImpl"/>
12   </component>
13    ...
14  <binding client="this.P1" server="Client.P2"/>
15  <binding client="Client.P3" server="Server.P4"/>
16  <binding client="Server.P5" server="this.P6"/>
17</definition>
```

LST. 2.1: Extrait de description Fractal

```
1 ...
2Component Client {
3  provides Runnable P2;
4  uses Service P3;
5  emits Event P5;
6}
7 ...
```

LST. 2.2: Extrait de description CCM

```
1<composite targetNamespace="http://helloworld" name="HelloWorld">
2  <service name="P1" promote="Client/P2">
3    <interface.java interface="java.lang.Runnable"/>
4  </service>
5  <component name="Client">
6    <service name="P2">
7      <interface.java interface="java.lang.Runnable"/>
8    </service>
9    <reference name="P3">
10     <interface.java interface="example.hw.Service"/>
11   </reference>
12   <implementation.java class="example.hw.ClientImpl"/>
13 </component>
14  ...
15 <wire source="Client/P3" target="Server/P4"/>
16</composite>
```

LST. 2.3: Extrait de description SCA

Architecture orientée services. Dans les architectures orientées services, la conception consiste à décrire les services avec les fonctionnalités qu'ils offrent, ainsi que la composition entre ces services, c'est-à-dire le flot de contrôle.

Dans le cas des services WEB, un service est spécifié à l'aide du langage de définition des services WEB (WSDL pour "*WEB Service Definition Language*") [W3C01]. La version 1.1 de ce langage a été définie en 2001 par le W3C. Ce langage contient les concepts propres au service WEB et permet une description succincte d'un service WEB, comme le montre l'exemple du listing 2.4. Ce langage permet de décrire les messages échangés (cf. lignes 4 à 6), le type du port du service WEB (cf. ligne 8) avec son opération (cf. ligne 9) et ses arguments en entrées (cf. ligne 10) et en sorties (cf. ligne 11), ainsi que le service WEB avec son port fourni (cf. lignes 15 à 19).

Pour décrire la composition des services WEB, l'architecte a besoin de raisonner sur le comportement de l'architecture. Afin de décrire cette composition, l'architecte peut utiliser la seconde version du DSL WS-BPEL ("*Web Services Business Process Execution Language*") qui a été définie en 2007 par le consortium OASIS [OAS07]. Ce langage reflète directement les concepts requis pour exprimer les interactions entre les services WEB en décrivant l'ordre d'enchaînement des invocations des autres services. Il met en avant deux grands concepts : le concept d'activité et le concept d'opérateur de flot de contrôle. Une activité correspond à une action de communication qui peut être une réception de messages, un envoi de messages ou une invocation d'un service WEB. Un opérateur de flot de contrôle permet de décrire l'ordre d'enchaînement des activités. Le

```
1<?xml version="1.0"?>
2<definitions name="Server" targetNamespace="...">
3  ...
4  <message name="namerequest">
5    <part name="body" element="xsd1:string"/>
6  </message>
7  ...
8  <portType name="P3Type">
9    <operation name="display">
10     <input message="tns:namerequest"/>
11     <output message="tns:result"/>
12   </operation>
13 </portType>
14  ...
15 <service name="Server">
16   <port name="P3" binding="tns:P3Binding">
17     <soap:address location="http://helloworld.com/server"/>
18   </port>
19 </service>
20  ...
21</definitions>
```

LST. 2.4: Extrait de description WSDL

```
1<process name="bankProcess" targetNamespace="..." xmlns=...>
2  <partnerLinks>
3    <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT" myRole="purchaseService" />
4     ...
5  </partnerLinks>
6  <variables>
7    <variable name="PO" messageType="lns:POMessage" />
8     ...
9  </variables>
10  ...
11  <sequence>
12    <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
13       operation="sendPurchaseOrder" variable="PO"
14       createInstance="yes">
15    </receive>
16    <flow>
17      <links>
18        <link name="ship-to-invoice" />
19        <link name="ship-to-scheduling" />
20      </links>
21      <sequence>
22        <invoke partnerLink="shipping" portType="lns:shippingPT"
23           operation="requestShipping"
24           inputVariable="shippingRequest"
25           outputVariable="shippingInfo">
26          <sources>
27            <source linkName="ship-to-invoice" />
28          </sources>
29        </invoke>
30         ...
31      </sequence>
32       ...
33    </flow>
34    <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
35       operation="sendPurchaseOrder" variable="Invoice">
36    </reply>
37  </sequence>
38</process>
```

LST. 2.5: Extrait de description BPEL

langage WS-BPEL repose sur les opérateurs de flot de contrôle structurés, incluant la séquence, l'alternative, l'itération et le parallélisme [RtHvdAM07]. Par exemple, le listing 2.5 est un extrait d'une description BPEL. La ligne 1 spécifie le nom du processus. Les lignes 2 à 5 définissent les dépendances requises vers les services WEB. Les lignes 6 à 9 décrivent les messages qui sont échangés entre les services. Les lignes 11 à 37 détaillent le flot de contrôle du processus. Les activités `receive` (lignes 12 à 15) et `invoke` (lignes 22 à 29) correspondent, respectivement, à l'attente de réception de messages provenant d'un service WEB et à l'envoi de messages vers un service WEB. Les activités de `sequence` (lignes 11 et 21) et `flow` (ligne 16) définissent, respectivement, un enchaînement séquentiel et parallèle des sous activités.

C) Ingénierie dirigée par les modèles

La seconde approche actuelle qui facilite la conception d'architectures logicielles est l'ingénierie dirigée par les modèles (IDM) [EVI05]. L'IDM repose aussi sur le principe d'abstraction pour concevoir un logiciel. Cette technologie a été mise en avant en 2000 par l'OMG avec leur spécification MDA (*"Model Driven Architecture"*) [Obj03a]. L'IDM a pour objectif de permettre la spécification d'un logiciel à un haut niveau d'abstraction en utilisant des modèles comme base d'abstraction.

L'IDM possède trois niveaux de conceptualisation, le niveau métamétamodèle, le niveau métamodèle et le niveau modèle. Pour définir l'application, l'architecte crée un modèle qui doit être conforme au métamodèle. Cette tâche est similaire à l'écriture d'une spécification en utilisant un DSL. Le rôle du métamodèle est similaire au rôle de la grammaire des DSLs. Il modélise le domaine de l'application. Il doit contenir les concepts du domaine et exprimer les relations entre ces concepts. Le métamétamodèle est un modèle spécifique pour la définition de métamodèle,

il contient donc tous les concepts et les relations entre ces concepts qui permettent de créer des métamodèles. Son rôle est similaire au rôle de la BNF (*"Backus-Naur Form"*) pour les DSLs. L'OMG a défini le MOF (*"Meta Object Facility"*) qui est un métamétamodèle réflexif, c'est-à-dire qu'il est possible de modéliser MOF avec MOF lui même [Obj03b].

Ainsi, dans le cas des architectures logicielles, le métamodèle reflète les concepts qui sont requis pour concevoir une architecture à composants ou orientée services. De cette manière, l'architecte peut concevoir son architecture en élaborant un modèle conforme au métamodèle. De plus, dans la mesure où le métamodèle prend aussi en compte les règles de composition liées à la plate-forme à composants ou orientées services, l'IDM garantit que si le modèle créé par l'architecte est conforme au métamodèle, alors l'architecture décrite satisfait les règles de composition. Dans le cas des architectures à composants, différents métamodèles qui explicitent la structure d'une architecture ont été définis. Par exemple, l'OMG a défini un métamodèle d'architecture à composants avec son Langage de Modélisation Unifié (UML) [Obj07b]. De même, la plate-forme d'exécution FrasSCAti repose sur un métamodèle de SCA qui reflète tous les concepts de SCA requis pour concevoir une architeture [SMF+09].

Du côté des architectures orientées services, l'IDM est utilisée pour abstraire les concepts requis afin de décrire le comportement de l'architecture, c'est-à-dire l'ordre d'enchaînement des invocations de services. Par exemple, l'architecte peut utiliser BPMN (BPMN pour *"Business Process Modeling Notation"*) pour décrire le comportement de son architecture [Obj07a]. BPMN est une notation graphique facilement compréhensible par tous les utilisateurs qui permet d'expliciter le flot de contrôle, comme le montre la figure 2.9. De manière similaire à WS-BPEL, cette notation repose sur deux concepts : les tâches et les opérateurs de flot de contrôle. Une tâche correspond à une activité, comme par exemple une invocation de service. Un opérateur de flot de contrôle exprime l'ordre d'enchaînement des tâches. Contrairement à WS-BPEL, BPMN se base sur des opérateurs de flot de contrôle non structurés, comme la séquence, la convergence et la divergence du flot de contrôle. Afin de faciliter l'utilisation de BPMN, des travaux se sont intéressés à élaborer un métamodèle qui contient les concepts de BPMN [9]. Ainsi, la description du comportement correspond à créer un modèle conforme au métamodèle.

De plus, l'IDM n'est pas limitée à la description de l'architecture. Elle peut aussi être utilisée pour gérer le processus de développement par raffinement successif de la modélisation. De plus, elle facilite aussi la mise en œuvre technologique, comme l'élaboration rapide d'outils et d'applications. Par exemple, dans le monde des services WEB, une manière de concevoir l'architecture consiste à faire une description BPMN, puis à la raffiner en une description WS-BPEL [OvdADtH05]. Une des implémentations de ces travaux repose sur l'IDM. Concrètement, la notation BPMN et le langage WS-BPEL sont modélisés sous forme de métamodèle, et le lien entre les deux descriptions est implémenté par une transformation de modèles [10].

[9] http ://www.eclipse.org/bpmn/
[10] http ://www.eclipse.org/bpel/developers/model.php

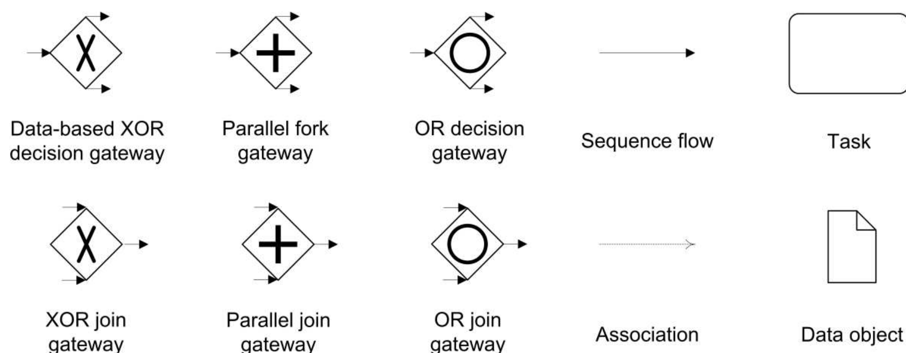

FIG. 2.9: Opérateurs de flot de contrôle de BPMN

D) Bilan

Lors de l'étape de conception l'architecte spécifie la composition des composants et des services qui constituent le logiciel. Dans le cas d'une architecture à composants, la composition est structurelle et dans le cas d'une architecture orientée services, elle est comportementale.

Afin de faciliter ce raisonnement, l'architecte dispose de deux approches : les ADLs et l'IDM. Ces deux approches reflètent directement les concepts requis pour la conception, grâce à un langage de description ou à un métamodèle. Ainsi l'architecte manipule les concepts qu'il connaît bien et dont la sémantique est clairement définie. Concrètement, dans le cas des ADLs, l'architecte spécifie son architecture de manière textuelle grâce à un langage dédié et dans le cas de l'IDM, il instancie un modèle conforme à un métamodèle. De plus, grâce à la forte sémantique des concepts, les ADLs et l'IDM permettent de vérifier si l'architecture décrite respecte les règles de composition du modèle. Ce support de vérification est internalisé dans le compilateur du langage dans le cas des ADLs. Dans le cas de l'IDM, les règles de composition peuvent être spécifiées avec un langage de contraintes, comme OCL (*"Object Constraint Language"*). Néanmoins, malgré ces similitudes, les ADLs sont utilisés exclusivement pour effectuer une description, alors que l'IDM peut aussi prendre en compte le processus de développement.

2.2.4 Etape d'implémentation

Durant l'étape d'implémentation, les développeurs codent le logiciel conformément à la spécification écrite par l'architecte. Pour une architecture à composants ou orientée services, l'implémentation équivaut à l'écriture du code des composants primitifs et des services. Cependant, bien que l'architecture logicielle permette un développement modulaire, son implémentation reste difficile à appréhender par les développeurs. En effet, pour chaque composant et service, en plus d'écrire le code métier, c'est-à-dire le code des fonctionnalités offertes par le composant ou le service, les développeurs doivent aussi écrire le code technique spécifique au modèle à composants utilisé ou à la plate-forme d'exécution des services. Ce code permet aux composants et aux services de communiquer entre eux. De plus, ce code technique est répétitif et doit être codé pour chaque composant et service constituant le logiciel. Mais aussi, selon le modèle à composants, ce code peut être verbeux et peut aussi contenir des informations redondantes avec celles contenues dans l'ADL, pouvant ainsi être source d'incohérence avec la spécification. Globalement, le code technique est entrelacé avec le code métier. De plus l'écriture de ce code est laborieux et source d'erreurs.

Par exemple, le listing 2.6 correspond à l'implémentation Fractal Julia du composant `Client` présenté dans la section 2.1.1. Dans ce code, seule la lignes 6 code le métier du composant, qui

```
1  public class ClientImpl implements Runnable, BindingController {
2    //interface serveur
3    private Service P3;
4    //code métier
5    public void run() {
6      this.P3.display("test");
7    }
8    //code technique
9    public String[] listFc () { return new String[] { "P3" }; }
10   public Object lookupFc (String name) {
11      if (name.equals("P3")) return this.P3;
12      return null;
13   }
14   public Object bindFc (String name, Object itf) {
15      if (name.equals("P3")) this.P3 = (Service)itf;
16   }
17   public Object unbindFc (String name) {
18      if (name.equals("P3")) this.P3 = null;
19   }
20 }
```

LST. 2.6: Implémentation Fractal du composant `Client`

est ici une invocation de l'interface requise `P3`. Tout le code restant est le code technique qui gère la création des liaisons du composant avec les autres composants. Au total, l'implémentation de ce composant est constituée de 95% de code technique.

Dans un souci de faciliter l'implémentation des composants et des services, quelques modèles à composants ou orientés services proposent des approches basées sur le principe de *séparation des préoccupations*. Ces approches visent à isoler le code technique du code métier. Nous détaillons deux grandes approches répandues : la programmation générative et la programmation par attribut.

A) Programmation générative

Avec cette approche, le code technique est généré à partir de la description de l'architecture, c'est-à-dire à partir de l'ADL ou du WSDL. Le modèle à composants CORBA est un des premiers modèles reposant sur cette approche. Les composants sont décrits avec l'IDL, indépendamment de tout langage de programmation. Ensuite un générateur de code spécifique à chaque langage de programmation génère tout le code technique dans ce langage. Ce code technique est clairement séparé du code métier écrit par les développeurs. Par exemple, dans le cas d'une implémentation Java, les codes technique et métier sont séparés dans des classes distinctes.

B) Programmation par attribut

La programmation par attribut vise à marquer certains éléments du code pour leur ajouter une sémantique liée à un domaine [Paw05]. Concrètement, ce marquage repose sur le mécanisme d'annotations fourni dans le langage de programmation. Son expression est donc très dépendante du niveau de support des annotations au sein de ce langage. Par exemple, Java supporte explicitement les annotations à partir de la version 5, auparavant les annotations devaient être écrites à l'intérieur des blocs de commentaires.

Dans le cas des modèles à composants, les annotations correspondent aux concepts du modèle à composants utilisé, comme le concept de port fourni et requis. La programmation par attribut permet donc une séparation claire entre le code métier et les informations techniques liées au modèle à composants. Des générateurs spécifiques à chaque modèle à composants analysent ensuite ces annotations afin de générer tout le code technique lié à ce modèle. En outre, les générateurs peuvent effectuer une vérification sémantique du code source grâce aux annotations et en déduire si le code respecte les règles du modèle à composants.

De plus en plus de modèles à composants récents utilisent la programmation par attribut pour faciliter l'implémentation des composants, comme le modèle à composants EJB3 [SUN06b], Fractal [BCS04] et SCA [BII+07]. Par exemple, le modèle à composants Fractal dispose de Fractal Annotation [Rou06]. Pour la plate-forme d'exécution Fractal reposant sur Java version 5, ces annotations sont des annotations Java5. Le tableau 2.2 présente quelques annotations. Pour chacune de ces annotations, il détaille sur quel élément du langage de programmation elle porte et sa signification. Avec ce mécanisme d'annotations, les interfaces Java correspondant à des ports

Annotation	Élément	Paramètres	Description
@Component	Classe Java	name provides uses	nom du composant liste des ports fournis liste des fichiers ADLs utilisés
@Interface	Interface Java	name signature	nom du port fournis signature de l'interface Java
@Requires	Champ Java	name cardinality contengency	nom du port requis cardinalité du port (unique ou collection) contingence du port (obligatoire ou optionnelle)

TAB. 2.2: Annotations Fractal

```
1 @Component(name="Client" , provides=@Interface(name="P2",signature=java.lang.Runnable.class))
2 public class ClientImpl implements Runnable {
3   //interface serveur
4   @Requires(name="P3")
5   private Service P3;
6   //code métier
7   public void run() {
8     this.P3.display("test");
9   }
10 }
```

LST. 2.7: Implémentation Fractal du composant `Client` avec les annotations

fournis sont annotées `@Interface`. Le nom du port est fixé par le paramètre `name` de l'annotation. Le listing 2.7 correspond à l'implémentation Fractal du composant `Client`. Comparée à la version sans annotation (cf. listing 2.6), la version avec annotation est 50% plus réduite. Le code technique se résume à uniquement deux lignes : une annotation `@Interface` à la ligne 1 pour décrire l'interface fournie `P2` et une annotation `@Requires` pour décrire l'interface requise `P3`. Avec les annotations, le code technique correspond à 20% du code du composant.

Les services WEB disposent eux aussi d'un mécanisme d'annotations en fonction du langage de programmation utilisé. SUN a défini un ensemble d'annotations pour les services WEB dans la JSR 181 [SUN06a] et les a inclues ultérieurement en standard dans Java à partir de sa version 6. De manière similaire aux annotations pour les composants, les annotations associées aux services WEB correspondent aux concepts de service, méthode des services et référence vers d'autres services (cf. tableau 2.3). Par exemple la classe d'implémentation d'un service est annotée `@WebService`. Le nom du service est spécifié avec le paramètre `serviceName` de l'annotation. Microsoft a défini, de même, des annotations pour les services WEB avec son langage C# [Mic07].

Annotation	Élément	Paramètres	Description
@WebService	Classe Java	serviceName targetNamespace	nom du service WEB espace de nommage
@WebMethod	Méthode Java		méthode du service
@WebServiceRef	Champ Java	wsdlLocation	adresse WEB du fichier de description du service requis

TAB. 2.3: Annotation pour les services WEB

C) Bilan

Lors de l'étape d'implémentation, les développeurs écrivent le code des composants et des services en fonction de la description de l'architecture effectuée par l'architecte. Ce code se compose d'une partie métier et d'une partie technique liée au modèle à composants ou orienté services utilisé. La programmation générative et la programmation par attribut simplifient l'implémentation des composants et des services en générant automatiquement une grande partie de ce code technique. La programmation générative se base directement sur la description de la conception décrite par l'architecte. En conséquence, elle garantit une cohérence forte entre la conception et l'implémentation. Avec la programmation par attribut, les développeurs spécifient à l'aide d'annotations les informations liées à l'architecture logicielle, comme les noms des ports requis et fournis de chaque composant. Néanmoins, dans la mesure où cette tâche est manuelle, les développeurs peuvent se tromper en recopiant les informations spécifiées par l'architecte. En conséquence, cette approche ne garantit pas la cohérence de l'implémentation avec la spécification établie durant la conception.

2.2.5 Etape de validation

L'étape de validation est l'étape clé pour élaborer un logiciel fiable. En effet, pour concevoir une architecture à composants ou orientée services, l'architecte logiciel doit respecter les règles de composition du modèle à composants ou orienté services utilisé. Ces règles servent à garantir que l'assemblage est conforme au modèle et qu'il peut en conséquence être déployé sur la plate-forme d'exécution associée au modèle.

De plus, afin de créer un logiciel fiable, l'architecte logiciel a aussi besoin de spécifier les exigences que le logiciel doit satisfaire. Une approche qui permet la description des exigences consiste à enrichir la description des composants et des services avec des propriétés applicatives. Il est donc ensuite nécessaire de valider que le logiciel résultant de la composition de composants et de services, ne viole pas les propriétés applicatives.

Les approches pour valider un logiciel se classent en deux grandes catégories : la validation statique et la validation dynamique. La validation statique vise à détecter toute violation des propriétés sans avoir besoin d'exécuter le logiciel. Elle correspond à effectuer des analyses statiques qui vérifient la description de l'architecture définie pendant l'étape de conception. La validation dynamique consiste à détecter les violations en exécutant le logiciel et en vérifiant que les données d'exécution respectent les propriétés applicatives. Elle est mise en œuvre par des analyses dynamiques.

Dans une première section, nous présentons comment un architecte spécifie ces exigences à l'aide des propriétés applicatives. Puis, nous détaillons les analyses statiques. Enfin, nous décrivons les analyses dynamiques.

A) Propriétés applicatives

Les propriétés applicatives permettent de décrire comment le composant ou le service interagit avec son environnement et comment son environnement peut agir sur lui. Elles expriment les contraintes de réutilisation que le service ou le composant impose sur son environnement, ainsi que les garanties qu'il offre. Beugnard et al. a défini une taxinomie des propriétés applicatives composée de quatre niveaux [BJPW99] :

- Le premier niveau correspond à un niveau basique. Il porte sur les propriétés syntaxiques des opérations des ports, c'est-à-dire le nom des opérations, le type des paramètres, les exceptions, etc.
- Le second niveau est appelé niveau comportemental. Il correspond à des propriétés qui peuvent être vérifiées par des pré-conditions, post-conditions et des invariants sur les opérations.
- Le troisième niveau correspond au niveau de synchronisation. Ce niveau décrit les interactions de l'entité avec son environnement. Ce niveau prend en compte l'exécution concurrente des entités.
- Le quatrième niveau correspond au niveau quantitatif. Il correspond aux propriétés non fonctionnelles du logiciel, comme les propriétés de performance, de maintenance ou de sécurité.

Dans notre approche, nous gardons ces quatre niveaux, mais nous affinons le découpage du second niveau de Beugnard et al. Nous prenons aussi en compte la sémantique des éléments qui sont contraints. Nous différencions donc les pré/post conditions et les invariants qui contraignent les éléments structurels de l'architecture, c'est-à-dire les composants, les services, les ports et les connecteurs, de ceux qui concernent les valeurs des messages échangés. Nous proposons de classer les propriétés applicatives selon les quatre niveaux suivants : structurel, comportemental, de flot de données et de qualité de service (QdS) :

Les propriétés structurelles permettent à l'architecte d'exprimer les contraintes structurelles sur l'assemblage constituant le logiciel. Ce sont des invariants qui portent sur les composants, les services, les ports et les connecteurs. Ce niveau correspond au premier niveau et à un sous ensemble du second niveau de Beugnard et al.

Les propriétés comportementales autorisent l'architecte à décrire le flot de contrôle entrant et sortant des ports de chaque composant et service. Elles sont similaires au troisième niveau de Beugnard et al. Nous avons cependant choisi de renommer le nom de ce niveau dans la mesure où le terme "comportement" est le plus répandu dans le domaine des architectures logicielles [All97, Mag99, PV02, RPU+07].

Les propriétés de flot de données permettent de restreindre les valeurs des données entrantes et sortantes des composants et des services. Elles correspondent à un sous ensemble du second niveau de Beugnard et al.

Les propriétés de qualité de services couvrent une large catégorie de propriétés non fonctionnelles incluant la performance, la sécurité et la disponibilité des composants et des services. Elles correspondent au quatrième niveau.

B) Analyse statique

Une fois que l'architecte a composé ses composants et ses services, il a besoin de vérifier si les composants et services qui interagissent entre eux satisfont les propriétés applicatives de l'architecture. Puisque l'architecture peut être large et complexe, il n'est pas envisageable que l'architecte analyse manuellement son architecture pour identifier des violations des propriétés applicatives. Donc, afin de répondre à ce besoin, quelques modèles et plates-formes sont associés avec des outils d'analyse statique de l'architecture qui valident les interactions entre les composants et les services, comme par exemple Wright [All97], Darwin [MDEK95] et SafArchie [Bar05].

Les travaux autour de SafArchie proposent de classer le résultat d'une analyse d'une interaction selon trois types : compatible, incompatible ou partiellement compatible :

- Il est **compatible** si les garanties offertes par les composants ou les services remplissent les conditions imposées par les composants ou les services assemblés entre eux.
- Il est **incompatible** quand au moins une des spécifications est violée.
- Il est **partiellement compatible** quand l'analyse ne déclenche aucune erreur mais ne peut pas être terminée car au moins une des propriétés dépend de données dont les valeurs ne peuvent être connues que lors de l'exécution du logiciel. Ce cas est typique de l'analyse du flot de données ou de la performance, comme le temps de réponse.

Néanmoins, la très grande majorité des outils d'analyse statique existants ne prennent pas en considération les interactions partiellement compatibles. Au contraire, ils considèrent, dans ce cas de figure, que l'interaction est incompatible, comme le fait par exemple l'outil d'analyse du comportement de Wright [Hoa85] et de Darwin [Mag99]. En conséquence, ces outils sont trop rigides et limitent la réutilisabilité des composants et des services. En effet, selon le contexte d'utilisation du logiciel, il est possible que l'interaction partiellement compatible ne viole aucune propriété applicative. Or les outils existants vont empêcher la conception de ce logiciel.

D'autres outils, comme SafArchie, notifient le problème potentiel à l'architecte, mais c'est ensuite à la charge de l'architecte de terminer manuellement l'analyse lorsque le logiciel s'exécutera [Bar05]. En conséquence, dans ce cas, l'architecte a besoin de décrire les analyses dynamiques afin de prendre en compte les données d'exécution. Cette étape manuelle est complexe et source d'erreurs. L'architecte a besoin d'avoir une maîtrise de l'implémentation des composants et des services afin de pouvoir y insérer l'assertion qui testera dynamiquement la violation de la propriété.

C) Analyse dynamique

Les analyses dynamiques correspondent à une procédure permettant la vérification partielle d'un logiciel, l'objectif étant d'identifier le maximum de comportements problématiques, c'est-à-dire que le comportement réel du logiciel est différent du comportement attendu et spécifié lors de la conception. Le comité international du test logiciel (ISTQB pour *"International Software Testing Qualification board"*) classe les tests selon quatre niveaux [MGF+07].

– Les tests unitaires visent à vérifier que chaque composant ou service a été implémenté correctement. Ce niveau de test est effectué après l'implémentation. Toutes les fonctionnalités offertes par les ports fournis du composant sont testées indépendamment dans un environnement simulé, c'est-à-dire que chaque port requis du composant est connecté avec un composant simulant le comportement attendu du composant. Ces tests peuvent être utilisés pour vérifier que l'implémentation du composant est conforme avec sa spécification contractuelle.
– Les tests d'intégration sont effectués après les tests unitaires. Ils visent à vérifier que le comportement de l'assemblage de composants est bien conforme aux spécifications. Il existe différentes stratégies [MGF+07]. Par exemple, la stratégie *"Big Bang"* teste directement l'assemblage complet des composants ou des services. La stratégie *"bottom-up"* teste d'abord les assemblages de composants de plus bas niveau hiérarchique, puis récursivement teste les composites de niveau hiérarchique supérieur. De manière similaire, la stratégie *"top-down"* commence par tester les composites de plus haut niveau hiérarchique puis teste récursivement les sous composants. L'ISTQB conseille l'utilisation d'une stratégie incrémentale afin de faciliter la localisation du problème.
– Les tests systèmes arrivent après les tests d'intégration. Ils visent à détecter toute incohérente entre le logiciel et le matériel ou les autres logiciels déjà existants.
– Le test d'acceptation, aussi appelé recette, vise à faire essayer le logiciel par les utilisateurs afin que ceux-ci signalent si le logiciel est conforme à leur attente.

Tous ces tests doivent être le moins possible intrusifs dans le sens où ils ne doivent pas modifier le comportement du logiciel, sinon ils faussent les tests. Dans notre approche, les analyses dynamiques correspondent aux tests d'intégration dans la mesure où nos travaux se focalisent sur l'analyse de la compatibilité des interactions entre les composants et les services.

D) Bilan

Afin d'augmenter la confiance que l'architecte peut avoir sur la fiabilité de son architecture, l'architecte a besoin d'avoir de pouvoir spécifier, lors de la conception, les diverses propriétés applicatives que l'architecture doit respecter, c'est-à-dire les propriétés structurelles, comportementales, de flot de données et de QdS.

La vérification de la satisfaction des propriétés applicatives repose sur deux grandes catégories d'analyses qui sont complémentaires : les analyses statiques et les analyses dynamiques. Les analyses statiques sont effectuées après la conception. Elles vérifient que l'architecture logicielle, décrite par l'architecte, satisfait les propriétés applicatives des composants et des services. Elles permettent d'identifier si les interactions entre les composants et les services sont compatibles, incompatibles ou partiellement compatibles. Quant aux analyses dynamiques, elles sont effectuées après le déploiement, durant l'étape de validation dynamique du logiciel. Elles identifient si des propriétés applicatives sont violées lors de l'exécution du logiciel.

Néanmoins, à notre connaissance, les approchent ne prennent pas en compte complètement la validation d'une interaction partiellement compatible. Soit elles considèrent l'interaction incompatible, soit elles signalent le problème à l'architecte. Or, dans le cas des interactions partiellement compatibles, les analyses ont besoin de données qui ne sont connues que lors de l'exécution du logiciel. En conséquence, dans ce cas, il est nécessaire d'effectuer une analyse dynamique de l'interaction, lors de l'exécution du logiciel, pour finir la validation. L'unification des analyses statiques et dynamiques est donc requise pour développer un logiciel fiable.

2.2.6 Etape de déploiement

L'étape de déploiement est réalisée par un administrateur système. Cette étape a pour but de mettre à disposition des utilisateurs le logiciel. Dans le cas des architectures à composants ou orientées services, le déploiement inclut l'installation de la plate-forme d'exécution et les composants ou les services constituants l'application. Différents travaux, comme DeployWare [Dub08] ou Jade [Sic08], proposent une approche permettant la description du système, des machines

distantes cibles, des plates-formes d'exécution et des composants applicatifs afin d'automatiser l'étape de déploiement. Dans le cadre de ce document, nous considérons que les plates-formes d'exécution sont déjà déployées et que, seuls les composants et services constituant le logiciel nécessitent un déploiement.

Le support de déploiement des composants et des services diffère suivant la plate-forme d'exécution utilisée. La majorité des plates-formes offre deux méthodes de déploiement : l'ADL et l'API.

A) Déploiement avec l'ADL

La méthode de déploiement par l'ADL consiste à déployer la totalité du système à partir de sa description ADL. Elle est effectuée par l'outil de déploiement associé à la plate-forme. Cette méthode est généralement utilisée pour le déploiement initial du logiciel.

B) Déploiement avec l'API

La méthode par l'API est une approche programmative. La plate-forme à composants expose une API de déploiement permettant de déployer et de replier des composants ou des services pendant l'exécution du logiciel. Avec cette méthode, l'administrateur système écrit un script qui contient la séquence des opérations nécessaires pour déployer le logiciel.

C) Bilan

Afin de mettre le logiciel à disposition des clients, l'administrateur système a besoin de déployer les composants et les services qui constituent le logiciel. Il dispose de deux méthodes pour effectuer cette tâche : la méthode avec l'ADL et la méthode avec l'API. La méthode de déploiement avec l'ADL est plus rigide que celle avec l'API dans la mesure où elle ne permet que de déployer un système complet. La méthode avec l'API autorise l'administrateur à déployer un sous ensemble de l'architecture. Néanmoins, cette seconde méthode est manuelle. En dépit de tout le soin apporté par l'administrateur pour écrire le script de déploiement, celui-ci peut introduire des erreurs. En conséquence, il est possible que le déploiement échoue ou que l'architecture déployée soit incohérente avec l'architecture décrite par l'architecte. D'un autre coté, puisque la méthode de déploiement avec l'ADL est entièrement automatique, elle garantit que le logiciel déployé correspond à la description de l'architecte. Globalement, pour empêcher l'introduction d'incohérence entre le logiciel déployé et la description de l'architecture, il est nécessaire d'automatiser l'étape de déploiement.

2.3 Conclusion

Dans le cadre de ce document, nous nous intéressons à la problématique de l'évolution fiable des applications à composants ou orientées services. Ce chapitre a donc présenté, dans un premier temps, les architectures logicielles, ainsi que l'étendue des différents modèles à composants et des modèles orientés services. Il a mis en lumière le fait que chaque modèle et chaque plate-forme d'exécution associée possèdent des avantages et des inconvénients selon le domaine de l'application qui doit être développée, comme par un exemple un système d'information ou un intergiciel. De plus chacun de ces modèles possède des règles de composition différentes que l'architecte doit satisfaire lorsqu'il conçoit son logiciel. Donc, afin de pouvoir exploiter un maximum de modèles et de plates-formes existants, nous souhaitons que notre canevas de développement soit multi plates-formes et qu'il prenne en compte la diversité des règles de composition.

Dans un second temps, ce chapitre a donné un aperçu du cycle de développement d'un logiciel. Il a mis en évidence que ce cycle est composé de cinq grandes étapes : les étapes d'analyse des besoins, de conception, d'implémentation, de validation et de déploiement. Pour chacune de ces étapes, ce chapitre a détaillé les grandes approches qui sont utilisées.

L'étape d'analyse des besoins est effectuée par un analyste qui rédige un cahier des charges spécifiant toutes les fonctionnalités que le logiciel doit offrir.

Lors de l'étape de conception, l'architecte logiciel modélise son architecture. Cette modélisation correspond à spécifier la structure de l'architecture dans le cas des architectures à composants, et à définir le comportement dans le cas des architectures orientées services. Deux approches facilitent la conception de l'architecture : les ADLs et l'IDM. Ainsi l'architecte peut décrire son architecture de manière textuelle grâce à un langage dédié ou alors en instanciant un métamodèle. De plus, afin de concevoir un logiciel fiable, l'architecte logiciel peut spécifier les propriétés applicatives qui sont requises et offertes par les composants et les services. Nous avons identifié quatre catégories de propriétés applicatives : les propriétés structurelles, comportementales, de flot de données et de QdS.

Durant l'étape d'implémentation, les développeurs écrivent le code des composants et des services. Cette étape se base sur le principe de séparation des préoccupations qui vise à séparer le code métier du code technique spécifique à la plate-forme d'exécution. Deux approches coexistent : la programmation générative qui permet de produire tout le code technique à partir de la description conceptuelle et la programmation par attribut qui permet d'isoler, au sein d'annotations dans l'implémentation des composants et des services, les informations de conception.

L'étape de validation est l'étape clé pour développer un logiciel fiable. Son rôle est de vérifier si le logiciel respecte les propriétés applicatives spécifiées par l'architecte. Afin de faciliter cette vérification, les plates-formes sont associées à des outils d'analyse statique et dynamique. Néanmoins, les approches de vérification existants ne sont pas complètement satisfaisantes. En effet, elles ne prennent pas en charge les interactions partiellement compatibles.

L'étape de déploiement est réalisée par un administrateur système. Lors de cette étape, l'administrateur déploie les composants et les services en fonction de la description de l'architecture faite par l'architecte.

Globalement, le développement d'un logiciel met en œuvre différents acteurs et repose sur plusieurs grands principes du génie logiciel : l'abstraction, la réutilisation et la séparation des préoccupations. Néanmoins, des incohérences peuvent être introduites lors du passage entre chaque étape du cycle, ce qui augmente la difficulté de créer un logiciel fiable. Nous souhaitons donc élaborer un canevas de développement logiciel qui réutilise les grands principes du génie logiciel, ainsi que les approches de développement existantes. De plus, afin de faciliter un développement fiable, notre canevas doit garder la cohérence entre chaque étape du cycle en se basant sur un couplage fort des étapes et doit prend en charge la validation des interactions partiellement compatibles.

Ce chapitre a mis en évidence que la méthode de développement agile est très bien adaptée pour faire évoluer un logiciel. En effet, cette méthode suggère l'utilisation d'un cycle de développement itératif et incrémental qui permet d'ajouter ou supprimer des fonctionnalités lors de chaque itération afin de suivre l'évolution du besoin des clients. Néanmoins cette approche est avant tout une méthodologie et ne propose donc pas de moyens pour réaliser ce type de développement. Aussi, dans le chapitre suivant, nous allons détailler différents travaux connexes autour de l'évolution du logiciel lors des étapes de conception, d'implémentation, de déploiement et d'exécution. De plus, dans la mesure où l'étape de validation est une étape clé pour développer un logiciel fiable, le chapitre suivant donne de plus un large aperçu des approches autour de la validation d'une architecture logicielle.

Chapitre 3

Etat de l'art sur l'évolution et la validation d'architectures logicielles

Sommaire

3.1 Gestion de l'évolution dans les architectures logicielles **40**
- 3.1.1 Étape de conception 40
- 3.1.2 Étape d'implémentation 42
- 3.1.3 Étape de déploiement 42
- 3.1.4 Étape d'exécution 43
- 3.1.5 Bilan 45

3.2 Gestion de la validation statique et dynamique **46**
- 3.2.1 Critères de comparaison 46
- 3.2.2 Propriétés structurelles 47
- 3.2.3 Propriétés comportementales 50
- 3.2.4 Propriétés de flot de données 53
- 3.2.5 Propriétés de QdS 54
- 3.2.6 Bilan 56

3.3 Gestion du cycle de vie : le projet FAROS **58**
- 3.3.1 Processus de développement 59
- 3.3.2 Métamodèle pivot 60
- 3.3.3 Retour d'expérience 62

3.4 Bilan de l'état de l'art **64**
- 3.4.1 Conclusion 64
- 3.4.2 Cahier des charges 64

Ce chapitre présente et évalue les principaux travaux connexes à ceux de la thèse. Il s'appuie sur des travaux autour de l'évolution et de la validation de logiciels à composants ou orientés services, lors des différentes étapes du cycle de développement du logiciel, c'est-à-dire de l'étape de conception jusqu'à l'étape d'exécution. Pour commencer, la section 3.1 décrit les travaux concernant l'évolution du logiciel. Puis, la section 3.2 présente les différents travaux adressant la validation du logiciel. Cette validation inclut l'analyse statique de la conception et l'analyse dynamique du logiciel. Finalement, la section 3.3 présente un exemple de canevas de développement axé sur la validation du logiciel.

3.1 Gestion de l'évolution dans les architectures logicielles

Un système logiciel a besoin d'évoluer pour prendre en compte de nouvelles fonctionnalités afin de suivre les nouveaux besoins des utilisateurs et les changements qui se produisent sur les plates-formes technologiques. Cependant, ajouter une nouvelle fonctionnalité dans un système logiciel est une tâche complexe et source d'erreurs, surtout dans le cas de fonctionnalités qui s'entrelacent. D'une manière générale, l'évolution du logiciel requiert une modification invasive du logiciel.

Cette section énumère différents travaux représentatifs qui s'intéressent à l'évolution des systèmes logiciels à composants ou orientés services. Ces travaux sont classés en fonction du moment dans le cycle de développement où ils s'appliquent. Les sections 3.1.1 à 3.1.4 décrivent, respectivement, les travaux portant sur l'évolution du logiciel lors de l'étape de conception jusqu'à l'étape d'exécution.

3.1.1 Étape de conception

Pendant l'étape de conception, le logiciel a besoin d'évoluer afin qu'il satisfasse les nouveaux besoins des utilisateurs. Cette évolution correspond à mettre à jour la description de l'architecture par l'architecte logiciel dans le but d'ajouter ou de supprimer des composants ou des services. Concrètement, l'architecte a besoin de modifier les fichiers de description ADLs ou les modèles de l'architecture selon l'approche utilisée (cf. section 2.2.3). Cependant, en dépit de tout le soin apporté à la réalisation de cette tâche, l'intervention humaine reste complexe et source d'erreurs. En réponse à ce besoin d'évolution de l'architecture logicielle, quelques travaux se sont focalisés sur cette problématique. Nous présentons, dans un premier temps, les travaux qui reposent sur les ADLs, puis nous décrivons quelques travaux autour des modèles.

A) Les ADLs

TranSAT (TranSAT pour *"Transformation for Software ArchiTecture"*) est un canevas qui permet l'intégration de nouvelles fonctionnalités dans une architecture logicielle, de manière fiable [Bar05, BLLD06]. Cette approche est inspirée des approches par aspects (AOP pour *"Aspect Oriented Programming"*) [KLM+97]. Elle se base sur un patron d'architecture logicielle et sur un tisseur qui permet d'intégrer automatiquement les patrons dans une description d'une architecture logicielle écrite avec l'ADL SafArchie. Un patron d'architecture décrit explicitement les informations requises pour intégrer une fonctionnalité donnée de manière fiable dans une architecture. Cela permet au patron d'être réutilisable. Un patron est composé de trois parties : (1) le nouveau plan, (2) le masque de points de jonction et (3) l'ensemble des règles de transformation. (1) Le nouveau plan est un assemblage de composants qui correspond à une fonctionnalité. (2) Le masque de points de jonction exprime les propriétés que l'architecture cible doit satisfaire pour intégrer la fonctionnalité. Il définit les attentes du patron relatives au lieu d'intégration de l'architecture cible qu'il vient modifier. (3) Les règles de transformation spécifient la liste des opérations de transformation qui doivent être effectuées par le tisseur pour intégrer la fonctionnalité, comme par exemple ajouter un nouveau composant.

Cependant TranSAT est dédié à l'ADL SafArchie. Afin de remédier à cette limitation, FIESTA (FIESTA pour *"Framework for Incremental Evolution of SofTware Architecture"*) généralise l'approche de TranSAT pour la rendre exploitable pour différents ADLs [WLD07b, WLD07a]. Cette approche repose sur la définition d'un ADL générique qui permet la représentation du patron d'architecture logicielle, indépendamment de l'ADL qui est utilisé pour décrire l'architecture cible. De plus, dans un souci de faciliter l'utilisation des règles de transformation par un architecte, un langage de plus haut niveau a été élaboré dans FIESTA.

Par exemple, le listing 3.1 présente le patron d'architecture logicielle de FIESTA correspondant à la fonctionnalité de cache. Les lignes 1 à 6 décrivent le nouveau plan. Le nouveau plan est un composant `Cache` qui possède un port fourni `P2` et un port requis `P3`. Les lignes 8 à 12 représentent le masque de point de jonction. Le masque exprime que pour ajouter

```
1<!-- Nouveau plan -->
2<component name="Cache">
3  <interface name="P2" role="server" signature="Send"/>
4  <interface name="P3" role="client" signature="Send"/>
5  <content class="CacheImpl"/>
6</component>

8<!-- Masque de point de jonction -->
9ComponentMask Cm1,Cm2;
10CommunicationPointMask Pn on Cm1 linkable with Cache.P2;
11CommunicationPointMask Pm on Cm2 linkable with Cache.P3;
12ConnectorMask{Pn,Pm};

14<!-- Règles d intégration -->
15modifyConnector from Cm1.Pn to Cache.P2 and from Cache.P3 to Cm2.Pm;
```

LST. 3.1: Patron d'architecture logicielle de cache

le composant `Cache`, le lieu d'intégration de l'architecture cible doit être constitué de deux composants connectés ensemble avec un connecteur. Les ports des deux composants doivent être compatibles avec les ports requis et fournis du composant `Cache`. La compatibilité est déterminée au moment de l'intégration de la fonctionnalité et dépend du modèle à composants cible. Les lignes 14 et 15 correspondent aux règles de transformation. La règle indique que le connecteur entre `Pn` et `Pm` doit être enlevé et remplacé par un connecteur entre `Pn` et `P2`, et un autre connecteur entre `P3` et `Pm`.

D'autres travaux se sont intéressés à porter l'AOP dans les architectures logicielles. L'objectif général est de capturer et d'encapsuler une nouvelle fonctionnalité dans un module appelé aspect. Ainsi, la nouvelle fonctionnalité peut être intégrée dans l'architecture via un mécanisme de tissage d'aspects. Ces travaux sur les aspects se classent en deux grandes familles : ceux pour les architectures à composants et ceux pour les architectures orientées services.

Dans le cas des architectures à composants, le tissage d'un aspect dans l'architecture correspond à une modification structurelle de l'architecture, comme dans les travaux DAOP-ADL [PFT03] et FAC [Pes07]. Concrètement, l'aspect correspond à un composant. Il est tissé dans l'architecture grâce à l'ajout d'un connecteur entre le composant d'aspect et un composant de l'architecture.

Dans le cas des architectures orientées services, un aspect correspond à une modification du comportement, c'est-à-dire du flot de contrôle de l'architecture. Ainsi, un aspect correspond à la description du flot de contrôle de la nouvelle fonctionnalité [BVJ+06]. L'aspect est tissé en insérant ce nouveau comportement dans le comportement de l'architecture. Par exemple [BVJ+06] et [CM07] proposent une extension du langage WS-BPEL pour y ajouter les concepts liés à l'AOP.

Néanmoins, toutes les approches présentées précédemment sont purement incrémentales. Elles ne prennent en charge que l'intégration de nouvelles préoccupations. Elles ne permettent donc pas de supprimer les fonctionnalités qui sont devenues obsolètes, ni de modifier une fonctionnalité existante. En conséquence ces travaux ne résolvent qu'un sous ensemble de l'évolution du logiciel, à savoir l'intégration de nouvelles préoccupations.

B) Les Modèles

Dans le domaine de l'IDM, l'évolution de l'architecture logicielle correspond à modifier les modèles. Afin de faciliter ces modifications, le concept d'aspect a été porté sur les modèles et a donné lieu à l'AOM (*"Aspect-Oriented Modeling"*). Dans ce domaine, la description conceptuelle du logiciel est décrite à l'aide de plusieurs modèles. Ces différents modèles sont ensuite composés afin de créer le modèle intégral du logiciel. Ainsi, avec cette approche, l'architecte peut ajouter une nouvelle fonctionnalité en définissant le modèle de la fonctionnalité et en composant

ce modèle avec le modèle du système [BFFR05]. Toute la logique de composition des modèles repose sur la sémantique des opérateurs de composition. Par exemple, [RGF+06], [KHJ06] et [KFJ07] ont, respectivement, définis des opérateurs pour composer des diagrammes de classes, des scénarios et des diagrammes de séquences. Néanmoins, ces approches ne prennent en compte que l'intégration et ne permettent donc pas de supprimer des fonctionnalités du logiciel.

3.1.2 Étape d'implémentation

Lors de l'étape d'implémentation, le logiciel a besoin d'évoluer afin d'ajouter de nouvelles fonctionnalités et de supprimer les fonctionnalités obsolètes. Cette évolution correspond à une modification du code source des composants et des services par les développeurs. Généralement le code des fonctionnalités est *mélangé* avec le code des autres fonctionnalités et il est *dispersé* dans la majeure partie du code du logiciel. On dit que le code est *entrelacé*. En conséquence, l'ajout d'une fonctionnalité est une tâche invasive.

De même, la programmation orientée aspect, qui a été introduite en 1997, permet de résoudre cette problématique en encapsulant dans un aspect tout le code d'une fonctionnalité [KLM+97]. De nombreux travaux se sont intéressés à étendre les langages de programmation pour y ajouter le support des aspects, comme par exemple pour le langage C++ [SGSP02] et Java [Tea09]. Ainsi, ce support aide les développeurs à faire évoluer le code des composants et des services. Concrètement, l'AOP s'articule autour de six concepts :

La base est le code du logiciel initial qui ne contient pas encore l'aspect.

Un aspect encapsule une fonctionnalité transverse. Il contient un ou plusieurs codes advices.

Le code advice implémente le comportement d'un aspect. Un code advice peut être tissé avant, après ou autour de l'élément de base.

Un point de jonction est un point dans le flot d'exécution du logiciel. Il correspond généralement à des invocations de méthodes ou des accès aux attributs d'une classe.

La coupe permet de designer les lieux d'intégration de l'aspect. Elle regroupe un ensemble de points de jonction.

Le tissage est l'action qui insère les codes advices de l'aspect dans le code de base. Cette action peut être effectuée statiquement à partir du code source du logiciel ou dynamiquement pendant l'exécution du logiciel.

L'exemple du listing 3.2 est un aspect écrit avec AspectJ, qui est le tisseur d'aspect de référence pour les programmes Java [Tea09]. Cet aspect ajoute des actions de trace. La ligne 2 définit le point de jonction. Le point de jonction correspond à l'exécution de la méthode `main` du programme Java. Les lignes 4 à 9 décrivent le code advice. L'aspect insère avant (ligne 4) et après (ligne 7) l'exécution de la méthode `main`, une trace dans la console Java (lignes 5 et 8).

```
1 public aspect Tracing {
2   private pointcut mainMethod () : execution(public static void main(String[]));
3   before () : mainMethod() {
4     System.out.println("> " + thisJoinPoint);
5   }
6   after () : mainMethod() {
7     System.out.println("< " + thisJoinPoint);
8   }
9 }
```

LST. 3.2: Exemple d'aspect

3.1.3 Étape de déploiement

Nous avons vu que le déploiement d'un système logiciel consiste à mettre à disposition des utilisateurs le logiciel en l'installant sur des machines. Or, chaque machine dispose de ressources

matérielles différentes, comme les ressources CPU, la capacité mémoire, etc. L'adaptation lors de l'étape de déploiement consiste à affecter les composants logiciels sur chaque machine en fonction des ressources offertes par la machine et des ressources requises par le composant logiciel.

Ce domaine de recherche est appelé la planification du déploiement. Tous ces travaux permettent à un administrateur système de modéliser la topologie de l'infrastructure matérielle et l'architecture logicielle [KK04, WS09]. L'infrastructure matérielle décrit les machines, les connexions réseaux entre les machines, ainsi que les ressources CPU, mémoire, etc. offertes par chacune des machines. L'architecture logicielle représente les différents composants logiciels avec leurs interconnexions, ainsi que les ressources requises par chacun des composants. Afin d'établir une affectation des composants logiciels sur chaque machine qui respecte les contraintes matérielles des composants, ces approches utilisent un solveur de contraintes.

3.1.4 Étape d'exécution

Pendant l'exécution, le logiciel a aussi besoin d'évoluer pour prendre en charge de nouvelles fonctionnalités et pour s'adapter aux modifications de l'environnement d'exécution, comme par exemple une augmentation de la charge CPU ou une diminution du débit réseau. Cette évolution ne doit pas altérer les composants et services du système qui ne sont pas impliqués par l'évolution. Ainsi, les autres fonctionnalités du système qui ne sont pas concernées par l'évolution, restent disponibles aux utilisateurs. Afin de réaliser cet objectif, les plates-formes à composants et orientées services réflexifs, comme Fractal, OpenCOM, OpenCCM et FraSCAti, permettent la modification de l'application pendant son exécution en autorisant l'ajout et la suppression de composants et de connecteurs.

L'évolution du logiciel pendant son exécution peut être réalisée de trois manières différentes : manuellement par un administrateur système, par les composants ou les services eux-mêmes, ou par l'infrastructure. (1) Avec la méthode manuelle, l'administrateur système écrit un script de reconfiguration qui contient la succession des opérations de modification de l'application, comme par exemple ajouter/retirer un composant/service/connecteur [PMSD07]. (2) Dans le deuxième cas, les composants ou les services de l'application invoquent directement les APIs de reconfiguration de la plate-forme, comme ceux de la plate-forme Fractal [BCS04]. (3) Dans le troisième cas, l'architecte écrit des règles d'adaptation qui décrivent comment l'architecture doit évoluer en fonction des modifications qui se produisent dans l'environnement du logiciel. Ces règles sont analysées par un intergiciel externe qui contrôle automatiquement l'évolution de l'architecture en appelant l'API de reconfiguration de la plate-forme. Ce dernier cas correspond au domaine de l'informatique autonome (*"Autonomic Computing"*) [KC03].

Le premier cas est une solution ad-hoc et source d'erreurs. Seul l'administrateur connaît toute la logique de l'évolution. Le deuxième cas n'est pas non plus satisfaisant dans la mesure où il y a un mélange des préoccupations entre le code métier de l'application et le code concernant l'évolution de l'application. De plus, cette solution n'est pas flexible dans la mesure où toute la logique d'évolution est figée dans le code du logiciel. En conséquence, cette solution ne gère pas les évolutions qui n'ont pas été prévues dès la conception. La troisième solution est plus flexible car elle sépare clairement les préoccupations métier des préoccupations d'évolution qui sont encapsulées dans les règles d'adaptation. De plus, l'architecte logiciel peut ajuster les règles d'adaptation durant l'exécution du logiciel, en fonction de l'évolution des besoins des utilisateurs. Il existe un très grand nombre de travaux qui se focalisent sur l'informatique autonome. Dans cette section, nous donnons un large aperçu des différences et des points communs entre ces travaux.

D'une manière générale, tous les travaux qui s'intéressent à l'informatique autonome reposent sur les travaux d'IBM [KC03]. Ces travaux suggèrent l'utilisation d'une représentation abstraite de l'architecture, sur laquelle il est possible de raisonner (cf. figure 3.1). Cette représentation est associée à une boucle de contrôle qui va diriger l'évolution de l'application en fonction des règles d'adaptation qui ont été définies par l'architecte. Cette boucle est composée de quatre étapes. L'étape d'observation (1) consiste à capturer les changements dans l'en-

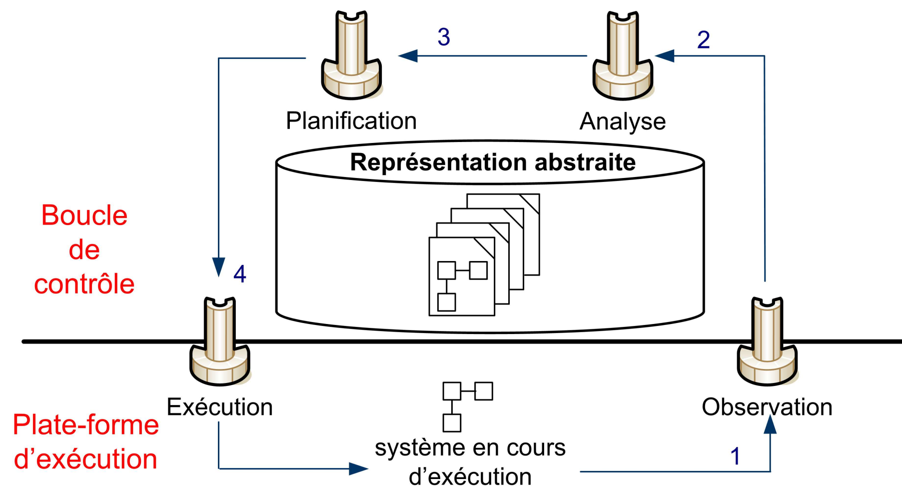

FIG. 3.1: Boucle de contrôle

vironnement de l'application, comme par exemple une modification du temps de réponse d'un composant ou d'un service. L'étape d'analyse (2) examine les changements qui ont eu lieu dans l'environnement afin de choisir la règle d'adaptation la plus appropriée. Par exemple un objectif de cette étape peut être de conserver le même niveau de qualité de l'application. L'étape de planification (3) prépare l'évolution en créant le script de reconfiguration. L'étape d'exécution (4) effectue l'évolution en exécutant le script de reconfiguration.

Néanmoins, les travaux autour de l'informatique autonome diffèrent sur trois points. D'une part, ils se spécialisent dans un domaine applicatif. D'autre part, ils se basent sur différents formalismes pour décrire l'architecture abstraite. Mais aussi, ils définissent de nouvelles boucles de contrôle.

Domaine applicatif. Les travaux concernant l'informatique autonome couvrent différents domaines, allant de l'auto-administration d'un système, comme par exemple Jade [Sic08] et DeployWare [Dub08], à l'auto-adaptation d'applications afin de s'adapter à un changement de contexte, telle qu'une modification du débit réseaux avec les travaux Rainbow [GCH+04] et Safran [Dav05]. Par exemple, Jade [Sic08] propose un mécanisme d'auto-réparation qui redéploie un serveur applicatif lorsque celui-ci devient défaillant.

Représentation abstraite. Dans la majorité des travaux existants, la représentation abstraite est purement structurelle, c'est-à-dire qu'elle décrit l'assemblage de composants du système. Seul le moyen mis en œuvre pour implémenter cette représentation diffère selon les travaux. Par exemple, Jade [Sic08] et DeployWare [Dub08] utilisent une architecture à composants Fractal. Rainbow [GCH+04] et les travaux de Brice et al. [MBNJ09] se servent de modèle et reposent sur l'AOM. Plastik [BJC05] utilise, quant à lui, une implémentation programmative ad-hoc.

Quelques travaux introduisent une représentation abstraite du comportement, c'est-à-dire du flot de contrôle de l'application. Par exemple, les travaux de Ji Zhang et Betty Cheng [ZC06] modélisent le comportement de l'application à l'aide de modèles et reposent sur l'AOM. Du coté des travaux autour des services WEB, VieDAME [MRD08] permet d'observer l'exécution d'un processus WS-BPEL et d'adapter le flot de contrôle en fonction des informations de QdS provenant de la plate-forme, comme le temps de réponse d'un service.

Boucle de contrôle. Il existe de très nombreux travaux qui se sont focalisés sur la boucle de contrôle, notamment sur les étapes d'observation et de planification. Concernant l'étape d'observation, des travaux comme WildCAT [DL05] et COSMOS [RCS08] proposent des solutions

```
1On (net_bandwidth = low) do {
2   detach MPEG-dec.req to conn-dec.p;
3   remove MPEG-dec;
4   Component H263-dec : decoder = new decoder extended with {
5       Property decoder-type = "H263";
6   };
7   Attachments
8           H263-dec.r to conn-dec.p;
9}
```

LST. 3.3: Exemple de règle de reconfiguration pour Plastik

permettant la capture des propriétés de QdS multidimensionnelles.

Concernant l'étape de planification, de nombreux travaux s'intéressent à définir différents paradigmes d'adaptation. Le paradigme ECA (événement, condition, action) [PSD98] est le plus utilisé. Par exemple, DynamicAcme [Wil01], Rainbow [GCH+04], Plastik [BJC05] et DeployWare [Dub08] effectuent une évaluation centralisée des règles ECA. Au contraire dans Safran, les règles ECA sont distribuées dans chaque composant du logiciel [Dav05]. Le paradigme d'adaptation peut aussi reposer sur la logique floue, comme par exemple dans [LN99] ou sur les fonctions d'utilités qui sont utilisées dans le projet Européen IST-MUSIC [RBD+09]. D'autres travaux, comme par exemple Dynaco [JBFAJLP06], n'imposent pas l'utilisation d'un paradigme d'adaptation, mais permettent, au contraire, de programmer la logique de l'adaptation dans les différents composants du logiciel. Parmi tous les paradigmes d'adaptation existants, le paradigme ECA reste le plus facile à utiliser par un architecte qui ne possède a priori aucune connaissance formelle dans les modèles mathématiques. Par exemple, le listing 3.3 est un exemple de règle de reconfiguration utilisée dans Plastik. Elle exprime le fait que lorsque la bande passante devient basse (cf. ligne 1), le composant décodeur MPEG doit être remplacé par un composant décodeur H263 (cf. lignes 2 à 8).

3.1.5 Bilan

Cette section a donné un large aperçu des différentes approches qui permettent de faire évoluer le logiciel afin de l'adapter aux nouveaux besoins des utilisateurs et aux changements se produisant dans l'environnement du logiciel. Toutes ces approches simplifient la tâche des différents acteurs, c'est-à-dire de l'architecte, du développeur et de l'administrateur. En effet, d'une part elles permettent d'isoler la nouvelle fonctionnalité séparément des autres fonctionnalités. Cette séparation prend différentes formes selon le moment, dans le cycle de développement, où l'évolution a lieu. Par exemple, l'évolution peut être encapsulée dans un patron d'architecture, dans un aspect ou dans une règle d'adaptation. D'autre part, l'intégration de la fonctionnalité dans le système est réalisée automatiquement et de manière transparente par une tisseur d'aspect ou la boucle de contrôle. Cependant, ces travaux ne permettent pas d'effectuer une modification radicale du logiciel, comme par exemple une suppression ou une modification des composants existants.

En outre, l'évolution du logiciel peut produire un logiciel incohérent, c'est-à-dire qui ne satisfait pas les propriétés applicatives exigées. Par exemple, après une évolution, le logiciel peut violer des contraintes structurelles. Très peu de travaux prennent en considération la fiabilisation des évolutions, c'est-à-dire qu'ils ne vérifient pas la validité du logiciel qui a évolué avant de mettre en œuvre l'adaptation. Nous donnons donc un aperçu des travaux qui portent sur la validation du logiciel dans la section 3.2.

D'autre part, l'évolution d'un logiciel met en jeu différents acteurs et est répartie sur tout le cycle de développement du logiciel, c'est-à-dire de l'étape de conception jusqu'à l'étape d'exécution. En effet, l'ajout d'une nouvelle fonctionnalité pendant l'exécution du logiciel implique, au préalable, que l'architecte ait conçu la fonctionnalité et que le développeur ait implémenté la fonctionnalité. Or, il est possible que des incohérences surviennent entre chacune des étapes du cycle de développement. Pour éviter ce problème, il est nécessaire de toujours

conserver un lien causal entre les éléments conceptuels spécifiés par l'architecte et les éléments d'exécution déployés dans la plate-forme. Pour illustrer ce besoin, la section 3.3 présente le canevas de développement FAROS, qui a pour objectif de permettre la propagation de contraintes applicatives de la conception jusque dans la plate-forme.

3.2 Gestion de la validation statique et dynamique

Après avoir conçu ou fait évoluer son architecture, il est nécessaire d'analyser l'architecture résultante afin de déterminer si toutes les propriétés applicatives désirées sont respectées. Il existe de très nombreux travaux qui s'intéressent à cette préoccupation. Afin de permettre l'exploitation de ces travaux dans le cadre de notre thèse, cette section établit une étude comparative des différents travaux ayant un lien avec l'étape de validation statique d'une architecture logicielle et dynamique du logiciel. Elle détaille les principaux travaux existants portant sur la spécification et l'analyse des quatre niveaux des propriétés applicatives, c'est-à-dire structurelle, comportementale, de flot de données et de QdS.

Pour commencer, la section 3.2.1 présente nos critères de comparaison. Ensuite les sections 3.2.2 à 3.2.5 détaillent quelques travaux représentatifs concernant la spécification et la validation des propriétés structurelles, comportementales, de flot de données et de QdS. Puis, la section 3.2.6 termine en dressant le bilan comparatif de toutes ses approches.

3.2.1 Critères de comparaison

Nous avons sélectionné trois critères de comparaison qui sont en lien avec la validation d'une évolution : les conditions d'utilisation des analyses, le support des analyses incrémentales et le niveau des messages d'erreurs produits. L'objectif de ces critères de comparaison est de déterminer l'utilisabilité de ces approches dans un cycle de développement itératif et incrémentale. Le but est d'identifier les informations requises et offertes par les différentes analyses afin de pouvoir élaborer notre cadre fédérateur qui permet l'intégration des outils d'analyse existants.

Condition d'utilisation des analyses. Afin de permettre la validation des évolutions, notre objectif est d'autoriser l'intégration d'un maximum d'outils d'analyse statique et dynamique existants. En conséquence, notre premier critère de comparaison concerne les conditions d'utilisation des analyses, c'est-à-dire qu'il vise à identifier les informations qui sont requises par les analyses. Ainsi notre objectif sera de capturer ces informations afin de les offrir aux outils d'analyse.

Incrémentale. Dans le cas d'une évolution du logiciel, seul un sous ensemble des composants ou des services est modifié. Les autres composants et services ne sont pas concernés par l'évolution, comme nous l'avons vu dans la section 3.1. Donc, dans un souci de maximiser les performances des analyses, quelques outils d'analyse existants prennent en charge les analyses incrémentales des propriétés applicatives. La raison principale de cette méthode d'analyse est d'augmenter la vitesse de l'analyse en ne prenant en compte que les modifications de conception effectuées par l'évolution afin de ne revalider que le strict minimum. Notre deuxième critère de comparaison vise à identifier si les outils d'analyse supportent ces analyses incrémentales. Dans le cas des analyses qui gèrent la vérification incrémentale, notre objectif sera aussi de déterminer quelles sont les informations supplémentaires qui sont requises par les analyses. Ainsi, nous pourrons exploiter les outils d'analyse incrémentale existants au sein de notre approche.

Niveau des messages d'erreurs. Dans les cas où l'outil d'analyse détecte des violations des propriétés applicatives, l'architecte a besoin de modifier sa conception pour corriger les problèmes. Pour faire cela, l'architecte doit déterminer l'origine du problème à partir des messages d'erreur qui sont fournis par l'outil d'analyse. Donc, afin de faciliter le développement, notre dernier critère de comparaison consiste à déterminer si les messages d'erreur produits pas les outils

d'analyse sont suffisamment de haut niveau d'abstraction pour être directement compréhensibles par l'architecte.

3.2.2 Propriétés structurelles

Les propriétés structurelles permettent à l'architecte de définir des contraintes structurelles limitant les connexions entre les composants ou les services. Cette section liste trois travaux portant sur la spécification et l'analyse des propriétés structurelles dans une architecture logicielle. Le premier travail est UML, un langage bien connu des architectes logicielles qui est dédié pour la conception de systèmes logiciels. Le second travail est Fractal, un modèle à composants qui est associé une plate-forme d'exécution légère. Le troisième est Acme, un ADL générique.

A) UML

UML est un langage de modélisation standardisé par l'OMG et bien connu des architectes logiciels [Obj07b]. Il est dédié pour la conception d'un système informatique. La version 2 d'UML fournit le métamodèle composant qui permet à un architecte logiciel de concevoir une architecture à composants. De plus, UML 2 est associé avec le langage de contraintes OCL (*"Object Constraint Language"*). Ce langage permet de définir des assertions structurelles dans un modèle UML lors de l'étape de conception. Il est apparu en 1997 avec la version 1.1 de UML [Obj97] et est passé en version 2.0 en 2006 [Obj06b]. Ce langage de contraintes a pour objectif d'être utilisable par des architectes n'ayant pas de connaissance, a priori, dans des techniques d'analyses formelles. Pour réaliser cet objectif, OCL fournit une syntaxe proche des langages impératifs, comme le C.

Une contrainte OCL est constituée de trois parties : le contexte, la portée temporelle et l'expression booléenne. (1) Le contexte sert à attacher la contrainte OCL à un élément du modèle, qui peut être une classe, une méthode, un attribut, etc., comme le montre la ligne 1 de l'exemple du listing 3.4. (2) La contrainte est complétée avec un stéréotype `inv`, `pre` ou `post` qui définit sa portée temporelle. Le stéréotype `inv` définit un invariant. Il s'applique sur une classe et concerne toute la durée de vie de cette classe. Les stéréotypes `pre` et `post` définissent respectivement des pré et des post conditions. Ils s'appliquent sur une méthode d'une classe et contraignent l'instant précédant ou succédant l'invocation de la méthode. (3) L'expression OCL est une expression booléenne qui contraint les éléments du modèle. Le mot clé `self` de l'expression correspond à l'élément du modèle qui porte la contrainte. Les notations *point* et *flèche* permettent de naviguer dans le modèle et d'accéder aux différents éléments, comme les attributs (cf. listing 3.4). La notation flèche est utilisée dans le cas des collections. OCL fournit aussi un ensemble d'opérateurs de manipulation de collection. Par exemple `forAll` et `exists` expriment, respectivement, une propriété universelle ou existentielle.

```
1 context Voiture
2   inv: self.roues->size()=4
```

LST. 3.4: Exemple de contrainte OCL

Condition d'utilisation des analyses. Bien qu'initialement OCL n'ait pas été conçu pour être exécutable, de nombreux travaux ont développé des outils d'analyse de contraintes OCL, comme par exemple EMF-OCL [1] ou [HDF00]. Tous ces outils basent leur validation sur une représentation statique du logiciel. Dans le cas d'une architecture logicielle, ils ont donc besoin de connaître la structure de l'application. Néanmoins, il est nécessaire de faire attention lors de l'intégration des outils existants dans la mesure où les différents outils existants sont incompatibles entre eux car ils n'interprètent pas le standard OCL de la même manière [GKB08].

[1] http ://www.eclipse.org/modeling/mdt/downloads/ ?project=ocl

Incrémentale. L'analyse incrémentale de contrainte OCL est très complexe. Seuls quelques travaux récents traitent ce problème, comme [CT06, Sag06]. Ces approches d'évaluation incrémentale de contraintes OCL se basent sur l'analyse des événements structuraux qui représentent une modification élémentaire du modèle, comme par exemple l'ajout d'une classe, la mise à jour d'un attribut ou l'insertion d'une relation. À partir de ces événements, cette approche détermine si les événements structuraux ne produisent pas un modèle qui va violer les contraintes OCL. Cette approche repose sur trois règles d'analyses : premièrement, une contrainte OCL n'est vérifiée que si les changements, qui ont eu lieu dans le modèle, peuvent induire une violation potentielle de la contrainte. Deuxièmement, les contraintes sont syntaxiquement écrites de la meilleure façon (à des fins de vérification incrémentale). Si nécessaire, les contraintes d'origine écrites par l'architecte logiciel sont remodelées afin qu'elles soient mieux adaptées à une analyse incrémentale. Troisièmement, les contraintes ne sont vérifiées que sur la partie du modèle qui a été modifiée, c'est-à-dire celles qui sont directement ou indirectement affectées par les événements structuraux.

Niveau des messages d'erreurs. Dans la mesure où les contraintes OCLs portent directement sur les éléments de conception, les outils de validation OCL fournissent des messages d'erreurs directement compréhensibles par l'architecte.

B) Fractal

Le modèle à composants Fractal, présenté dans la section 2.1.2, ne fournit pas à la base de mécanisme autorisant la spécification de propriétés structurelles [BCS04]. Cependant, la plate-forme d'exécution ConFract [2], développée par l'équipe I3S de l'Université de Sophia Antipolis et FT R&D, permet à l'architecte de définir des contraintes structurelles à l'aide du langage de contraintes CCLJ (invariants et pré/post conditions) [CMOR07]. La syntaxe de ce langage est inspirée d'OCL, mais a été adaptée au modèle à composants Fractal. CCLJ possède deux différences par rapport à OCL : le contexte et la portée architecturale. Une contrainte CCLJ est toujours attachée à une opération contenue dans un port d'un composant, comme le montre la ligne 1 du listing 3.5. Une contrainte CCLJ possède aussi une portée architecturale, c'est-à-dire que les éléments de l'architecture qui peuvent être référencés dans les contraintes sont restreints. CCLJ définit trois catégories de portée architecturale :

- Les *contraintes d'interface* sont limitées à la portée de l'interface, c'est-à-dire qu'elles ne peuvent référencer que les autres opérations du port qui contiennent l'opération qui porte la contrainte.
- Les *contraintes externes* portent sur les interfaces du composant. Elles ne peuvent référencer que les autres opérations contenues dans les ports du composant.
- Les *contraintes internes* ont une portée limitée au contenu d'un composant composite. Elles ne peuvent contraindre que les interfaces externes des sous-composants.

```
1 on <Voiture> context
2   inv: self.roues->size()=4
```

LST. 3.5: Exemple de contrainte CCLJ

Condition d'utilisation des analyses. Comme pour OCL, l'évaluation des contraintes CCLJ nécessite de connaître la structure de l'architecture.

Incrémentale. ConFract ne fournit pas de méthode d'analyse incrémentale.

[2] http ://www.i3s.unice.fr/~collet/confract.html

Niveau des messages d'erreurs. Les messages d'erreurs produits sont de haut niveau et portent sur la description de l'architecture.

C) Acme

Acme est un ADL générique qui permet de représenter la structure d'une architecture à composants indépendamment d'une plate-forme existante [GMW00]. Acme ne permet pas la spécification de propriétés structurelles. Cependant, une extension, nommée Armani [Mon01], autorise la spécification de contraintes structurelles. Ce langage de contraintes Armani se base sur des prédicats de logique de premier ordre. Il permet de définir des prédicats qui doivent rester vrais durant toute la vie du système, comme le montre le listing 3.6. Ce langage est similaire à OCL, mais il fournit en plus un ensemble d'opérations de manipulation spécifique pour les architectures logicielles. Ces opérations facilitent la définition de contraintes qui portent sur des concepts comme les interactions entre composants, la conformance de type ou les relations d'héritage.

```
1 Component Voiture
2 {
3   Port roues;
4   Invariant size(self.roues) = 4;
5 }
```

LST. 3.6: Exemple de contrainte Armani

Condition d'utilisation des analyses. L'environnement de développement AcmeStudio implémente un outil d'analyse des contraintes Armani [SG04]. Cet outil a besoin de connaître la structure de l'architecture.

Incrémentale. Armani ne prend pas en charge les analyses incrémentales de l'architecture. À chaque vérification du système, la totalité des contraintes Armani est revérifiée.

Niveau des messages d'erreurs. L'outil de validation est directement intégré dans l'environnement de développement AcmeStudio. Cela permet à l'outil de fournir des messages d'erreurs, de haut niveau d'abstraction, en relation avec la conception de l'architecture. De cette manière, l'architecte logiciel peut directement comprendre l'erreur et modifier sa conception.

D) Bilan

Il existe de nombreux travaux autour de la validation des propriétés structurelles de l'architecture du logicielle. Parmi tous ces travaux, OCL est un langage de contraintes bien connu des architectes logiciels. Il est applicable sur toutes les sortes de modèle. Néanmoins il n'est pas dédié pour spécifier des contraintes sur des modèles à composants ou orientés services. C'est pour cette raison que différents travaux se sont inspirés du langage OCL pour définir des langages de contraintes dédiés aux modèles à composants. Ces langages ajoutent des constructions de langage pour simplifier l'écriture des contraintes. Tous les outils de vérification associés à ces langages ont besoin de connaître la structure de l'architecture et, dans le cas des outils d'analyse incrémentale, ils requièrent en plus les informations de modifications structurelles. La validation de propriétés structurelles concernent principalement les modèles à composants dans la mesure où la structure est explicite, contrairement aux modèles orientés services qui reposent sur la description du comportement.

3.2.3 Propriétés comportementales

Les propriétés comportementales permettent à l'architecte logiciel de décrire le flot de contrôle entrant et sortant des composants et des services. Cette section présente quatre travaux principaux permettant la spécification et l'analyse de propriétés comportementales. Ces travaux ont été choisi car ils sont représentatifs de la large diversité des analyses existantes, que ce soit au niveau des conditions d'utilisabilité ou du support de l'incrémentabilité.

A) Wright

Wright est un modèle à composants qui permet la spécification des propriétés comportementales [All97]. Les propriétés comportementales sont décrites à l'aide du langage CSP (CSP pour *"Communicating Sequential Process"*) [Hoa85]. Ce langage repose sur l'algèbre de processus. Chaque composant est représenté par un processus. Un processus est un système dont le comportement est décrit sous forme d'une composition d'actions discrètes. Une action correspond soit à l'envoi d'un message, soit à la réception d'un message. Ces deux types d'actions reposent sur le mécanisme de rendez-vous, c'est-à-dire que l'action de réception se bloque en attente du message. De même, l'action d'envoi émet un message et attend qu'il soit consommé par une action de réception. La composition des actions est exprimée à l'aide d'un ensemble d'opérateurs tels que la séquence, le parallélisme ou l'alternative.

L'outil d'analyse prend en charge l'analyse des propriétés de sûreté et de vivacité du système. Les propriétés de sûreté garantissent que rien de mauvais ne va se produire dans l'assemblage de composants. Cette analyse vérifie que l'assemblage ne contient pas d'interblocage et que l'ordonnancement des messages spécifiés par l'architecte est respecté. Les propriétés de vivacité garantissent que quelque chose de bon peut toujours arriver.

Condition d'utilisation des analyses. L'outil d'analyse de Wright est purement statique. Il n'a donc besoin que de connaître la structure de l'architecture et le comportement de chaque composant. En conséquence, il ne peut identifier que deux types de compatibilité entre les composants : soit l'assemblage est correct soit il est incorrect. En effet, Wright peut identifier un assemblage comme étant incorrect même s'il est possible qu'aucune erreur ne survienne pendant l'exécution du système, c'est-à-dire que le chemin du flot de contrôle menant à l'erreur n'est jamais exécuté.

Incrémentale. L'outil d'analyse de Wright n'effectue pas d'analyse incrémentale. Il revérifie chacune des contraintes lors de chaque analyse.

Niveau des messages d'erreurs. Dans la mesure où l'analyse est effectuée sur une description de l'architecture du système, l'outil d'analyse produit des messages d'erreur directement en lien avec la conception. Ces messages sont donc directement compréhensibles par l'architecte logiciel.

B) SafArchie

SafArchie est un modèle à composants dédié pour la construction de logiciels fiables [Bar05]. SafArchie est associé au langage SFSP (SFSP pour *"Simple Finite State Process"*) qui permet la définition des propriétés comportementales. SFSP est issu d'un sous ensemble du langage FSP (FSP pour *"Finite State Process"*) [Mag99] qui sert à décrire les propriétés comportementales dans le modèle à composants Darwin [MDEK95]. Tout comme CSP, SFSP repose sur l'algèbre de processus. Il permet d'exprimer le comportement des composants sous forme de LTS (LTS pour *"Labeled Transition System"*). Le comportement est décrit sous forme de compositions d'actions. Les actions correspondent à un appel ou à la réception d'un appel de méthode et les opérateurs de composition sont la séquence, l'alternative, l'itération et le parallélisme. Par exemple, le listing 3.7 est une spécification comportementale très simple. Elle spécifie que le composant `cmp1` attend la réception d'un message (notation '`?`'), puis en séquence (notation '`;`') envoie (notation '`!`') une réponse. La notation '`$`' exprime que le message correspond à une réponse.

```
1AA=(?cmp1.p1.connect ; !cmp1.p1.connect$)
```

LST. 3.7: Exemple de spécification SFSP

Condition d'utilisation des analyses. L'outil d'analyse de SafArchie a besoin de connaître la structure et le comportement de chaque composant afin de vérifier des propriétés de sûreté. L'analyse est purement statique. Cependant, contrairement à Wright, l'outil définit trois types de compatibilité entre les composants : la compatibilité totale, la compatibilité partielle et l'incompatibilité. L'assemblage est totalement compatible si l'outil ne détecte aucune erreur. Il est partiellement compatible si l'outil détecte au moins un interblocage et qu'il existe des chemins du flot de contrôle qui évitent les interblocages. Il est incompatible si l'assemblage engendre obligatoirement un interblocage. Cette approche est donc beaucoup plus flexible pour l'architecte et lui permet de concevoir des assemblages partiellement compatibles. Cependant, l'outil n'effectue aucune analyse dynamique pour garantir que les chemins du flot de contrôle, conduisant aux interblocages, ne sont pas empruntés durant le fonctionnement du système. C'est donc à la charge du développeur de prendre en charge les analyses dynamiques en ajoutant, par exemple, manuellement des tests dans le code du système.

Incrémentale. SafArchie est associé avec TranSAT qui permet d'intégrer de nouvelles préoccupations, de manière incrémentale, dans une architecture SafArchie [Bar05, BLLD06]. L'outil d'analyse a donc été adapté afin de permettre une vérification incrémentale pour garantir que l'intégration d'une nouvelle préoccupation n'engendre pas un interblocage. Afin d'effectuer cette analyse, l'outil a besoin de connaître les opérations de modification de la structure de l'assemblage, comme par exemple l'ajout d'un nouveau composant. À partir de ces informations, l'outil identifie le composite qui nécessite une analyse des propriétés comportementales. Si le nouveau comportement du composite n'est pas équivalent à son ancien comportement, alors de manière récursive, l'outil recalcule le comportement du composite parent.

Niveau des messages d'erreurs. L'outil produit des messages de haut niveau d'abstraction. Il génère des messages d'erreur dans le cas où les interactions sont incompatibles afin que l'architecte corrige le problème et il produit des avertissements dans le cas où elles sont partiellement compatibles.

C) Fractal

De base, le modèle à composants Fractal ne fournit pas de support pour la spécification de propriétés comportementales. Néanmoins, la plate-forme d'exécution Fractal-BPC [3] permet à l'architecte logiciel de définir des propriétés comportementales [BABC+09]. Le support de ces propriétés résulte du portage des protocoles comportementaux du modèle à composants SOFA [KT02, PV02] pour le modèle à composants Fractal. Les protocoles comportementaux reposent aussi sur les algèbres de processus. Le comportement des composants est décrit sous forme de composition d'action. Comme pour SafArchie, les actions correspondent à un appel ou à la réception d'un appel de méthode. Les opérateurs de composition sont la séquence, l'alternative, l'itération et le parallélisme.

Condition d'utilisation des analyses. L'outil d'analyse de Fractal-BPC analyse statiquement la compatibilité comportementale de l'assemblage de composants, c'est-à-dire qu'il vérifie si l'assemblage contient un interblocage. Il a donc besoin de connaître la structure de l'architecture et le comportement de chaque composant. Cependant, comme pour Wright, l'analyse ne définit que deux types de compatibilité entre les composants : soit l'assemblage est correct soit il est

[3] http ://fractal.ow2.org/fractalbpc/index.html

incorrect. L'outil interdit donc à l'architecte logiciel de concevoir une architecture partiellement compatible. D'autre part, l'outil permet l'analyse dynamique de la conformité des composants par rapport à leur spécification. Pour réaliser cet objectif, la plate-forme Fractal-BPC intercepte les appels de méthode émis et reçus par chaque composant et vérifie que ces appels respectent la spécification des composants. Cette analyse requiert donc l'observation et la capture des communications entre les composants qui ont lieu dans la plate-forme.

Incrémentale. Fractal-BPC ne supporte pas la vérification incrémentale des propriétés comportementales. Il est donc nécessaire de revérifier la totalité du système à chaque modification de la conception.

Niveau des messages d'erreurs. L'outil produit des messages de haut niveau d'abstraction directement compréhensibles par l'architecte logiciel.

D) Services WEB

Les services WEB permettent la spécification du comportement, soit avec la notation graphique BPMN (BPMN pour *"Business Process Model and Notation"*) [Obj07a] ou soit avec le langage d'exécution WS-BPEL [OAS07], comme cela a été expliqué dans la section 2.2.3. Il existe une très grande variété de travaux qui analysent le comportement d'une architecture orientée services à partir d'une description BPMN ou BPEL.

Condition d'utilisation des analyses. Une grande partie des travaux existants porte sur l'analyse statique du comportement. Ces travaux utilisent principalement une description BPEL, comme [FUMK04] et [VvdA05], ou une description BPMN, comme [RPU+07]. Tous ces travaux transforment la description dans un autre formalisme pour permettre son analyse formelle. Par exemple, [FUMK04] transforme le BPEL en FSP, [VvdA05] transforme le BPEL en un réseau de Petri et [RPU+07] transforme le BPMN en un réseau de Petri.

D'autre travaux, comme [BGP08], effectuent des analyses dynamiques. Ces analyses reposent sur l'observation des échanges de messages entre les services pendant l'exécution du système.

Plus rarement, des travaux comme [RPG06], combinent une analyse statique avec une analyse dynamique. Cependant ces analyses sont découplées. En effet, l'analyse statique vérifie les propriétés de sûreté de l'architecture et l'analyse dynamique vérifie la conformance des services. Par exemple, dans [RPG06], l'analyse statique consiste à transformer la description BPEL en une expression événementielle afin de permettre son analyse par une approche d'analyse d'événements [KS86]. Cette analyse détecte les problèmes comme par exemple les interblocages et les chemins du flot de contrôle qui ne sont jamais utilisés. Cependant, comme pour Wright, cette analyse est rigide et ne définit que deux types de compatibilité entre les services : soit l'assemblage est correct, soit il est incorrect. En conséquence, cette approche n'autorise pas la conception d'un assemblage qui contient un interblocage, même si le chemin d'exécution menant à l'interblocage ne sera jamais emprunté dans le contexte d'utilisation du logiciel. L'analyse dynamique, quant à elle, repose sur l'observation des échanges de messages entre les services, pendant l'exécution des messages. Elle vérifie si ces échanges de messages respectent bien les propriétés comportementales. Néanmoins, l'analyse dynamique est indépendante de l'analyse statique, c'est-à-dire que le résultat de l'analyse statique n'est pas utilisé pour affiner l'analyse dynamique.

Incrémentale. À notre connaissance, il n'existe pas d'approche proposant une analyse incrémentale des propriétés comportementales pour les services WEB.

Niveau des messages d'erreurs. Les travaux reposant sur l'analyse des diagrammes BPMN fournissent des messages d'erreur de haut niveau directement compréhensibles par l'architecte.

Ils indiquent à l'architecte l'élément du diagramme qui est la source de la violation de la propriété. Au contraire, les travaux qui analysent le BPEL produisent des messages d'erreur de plus bas niveau. En effet, ils référencent généralement la ligne dans le fichier BPEL qui a provoqué la violation.

E) Bilan

Il existe de très nombreux langages différents permettant la spécification des propriétés comportementales. La majorité de ces langages reposent sur l'algèbre de processus ou sur les automates. Cependant, chacun de ces langages est fortement couplé avec un modèle à composants ou orienté services. En conséquence le choix par l'architecte d'un langage de spécification impose l'utilisation du modèle à composants ou orientée services associé. Une solution qui a été entreprise par certains travaux pour pallier ce problème de couplage a consisté à réécrire les outils d'analyses pour les porter sur d'autres modèles. Néanmoins cette solution nécessite un laborieux travail d'ingénierie.

D'autre part, tous ces travaux ont besoin de connaître la structure de l'architecture et le comportement de chaque composant et service. Dans le cas des services WEB, le flot de contrôle de l'application, représenté par une description BPMN ou BPEL, est aussi requis. Quant aux analyses incrémentales, elles ont besoin de connaître, en plus, les modifications structurelles qui se produisent durant l'exécution du logiciel. De plus, afin de supporter les analyses dynamiques du comportement, il est nécessaire de capturer les communications entre les composants et les services qui ont lieu dans la plate-forme.

3.2.4 Propriétés de flot de données

Les propriétés de flot de données permettent à l'architecte de spécifier des contraintes sur la valeur des messages échangés entre les composants et les services. Cette section détaille deux travaux principaux traitant la spécification et l'analyse des propriétés de flot de données dans les architectures logicielles.

A) Fractal

Le modèle à composants Fractal ne supporte pas, de base, la spécification des propriétés de flot de données. Cependant, la plate-forme ConFract, déjà présentée dans la section 3.2.2, ajoute ce support. Les propriétés de flots de données sont exprimées sous forme de pré et de post conditions booléennes qui portent sur les opérations contenues dans les ports des composants.

Condition d'utilisation des analyses. L'analyse de ces propriétés dans ConFract est dynamique. Elle nécessite la capture des données échangées entre les composants lors d'un appel ou de la réception d'un appel de méthode. Cette capture est réalisée dans la plate-forme à l'aide d'un mécanisme d'intercepteurs. La plate-forme teste ensuite si ces données ne violent pas les propriétés applicatives définies par l'architecte.

Incrémentale. L'analyse n'est pas incrémentale. Seuls des tests dynamiques sont effectués durant l'exécution du système.

Niveau des messages d'erreurs. Les messages d'erreur produits proviennent des analyses dynamiques et sont donc au niveau de l'implémentation. Cela signifie qu'ils sont compréhensibles par un développeur, qui a alors pour rôle de les faire remonter à l'architecte.

B) Acme

Nativement, Acme ne prend pas en charge la spécification et l'analyse des propriétés de flot de données. Cependant, l'extension Armani, déjà présentée dans la section 3.2.2, ajoute ce support. Les propriétés de flot de données sont définies avec des pré et des post conditions exprimées sous forme de prédicat de la logique du premier ordre.

Condition d'utilisation des analyses. Armani ne fournit pas de mécanisme de vérification des propriétés de flot de données. Cependant, dans [JRMWC07], les auteurs proposent une analyse exclusivement dynamique. Les contraintes de flot de données sont directement transformées en assertions exécutables dans le code du système. Cela signifie que les analyses dynamiques sont mélangées avec le code métier des composants de l'application.

Incrémentale. À notre connaissance, il n'existe pas de travaux qui traitent la vérification incrémentale des propriétés de flot de données dans un modèle à composants Acme.

Niveau des messages d'erreurs. L'approche dynamique proposée par [JRMWC07] fournit des messages d'erreur au niveau de l'implémentation. Ces messages indiquent le numéro de ligne dans le code source des composants qui correspond à l'assertion exécutable qui a été violée. En conséquence, les messages ne sont pas directement compréhensibles pas l'architecte. L'architecte a besoin de communiquer avec les développeurs pour identifier la propriété applicative qui a été violée.

C) Bilan

Le support des spécifications de flot de données est très peu répandu dans les modèles à composants ou orientés services. Il repose sur la spécification de pré et de post conditions qui contraignent la valeur des messages échangés. Néanmoins, ces approches sont exclusivement dynamiques. Elles nécessitent une observation des données qui sont échangées entre les composants ou les services. À notre connaissance, aucune approche ne propose une analyse statique de flot de donnée dans les modèles à composants, alors que ce support est très répandu dans le domaine de la validation partielle de programme [Kil73]. Ce support, dans les modèles à composants, permettrait d'informer l'architecte, dès l'étape de conception, si des propriétés applicatives sont violées. Il serait donc intéressant de permettre aussi l'intégration des travaux provenant de la validation partielle de programme.

3.2.5 Propriétés de QdS

Les propriétés applicatives de QdS servent à définir des contraintes extra-fonctionnelles, comme par exemple des contraintes concernant la performance ou la sécurité. Cette section détaille trois travaux portant sur la spécification et l'analyse des propriétés de QdS. Le premier travail concerne les modèles à composants, le second est une approche intégré dans une plateforme d'exécution et le troisème porte sur les services WEB.

A) Modèles à composants

CQML (CQML pour *"Component QoS Modeling Language"*) est un langage permettant d'exprimer des propriétés de QdS dans différents modèles à composants [TDG09]. CQML reprend les concepts de QML [FK98] et les étend pour les modèles à composants. CQML repose sur trois concepts clés : la caractérisation de la propriété de QdS, le contrat et le profil. (1) La caractérisation de la propriété de QdS permet de définir le domaine de valeur d'une propriété de QdS. Ce domaine est un intervalle ou un ensemble de valeurs. Les valeurs peuvent être, par exemple, des valeurs entières ou réelles. (2) Le contrat regroupe l'ensemble de contraintes de QdS portant sur

la propriété de QdS. L'expression d'une contrainte est une équation numérique. (3) Le profil associe un contrat avec une entité architecturale, comme par un exemple un composant. Il permet de définir les garanties de QdS offertes par le composant et les propriétés de QdS requises par le composant.

Condition d'utilisation des analyses. L'outil de validation a besoin de connaître la structure de l'architecture et les propriétés de QdS. Ces propriétés sont exprimées sous forme de garantie et de besoin, ce qui permet la validation statique du système. Toutefois, CQML ne met en œuvre qu'une analyse locale de la compatibilité entre une propriété requise et une propriété offerte. Ainsi, l'architecture est déclarée consistante si toutes les garanties offertes par un composant satisfont toutes les propriétés de QdS requises par les autres composants interagissant avec lui.

Incrémentale. CQML ne propose pas de mécanisme de vérification incrémentale.

Niveau des messages d'erreurs. Les messages d'erreur fournis par CQML portent sur les éléments de l'architecture et sont donc directement compréhensibles par l'architecte logiciel.

B) CCM

QuO est une approche qui permet de prendre en charge les propriétés de QdS dans une application CORBA [PLS+00]. Cette approche fournit le langage CDL (CDL pour *"Contract Description Language"*) dédié à la spécification des propriétés de QdS. Ce langage permet à un architecte de définir des états de QdS, c'est-à-dire un ensemble de domaines de valeurs de QdS. Il autorise aussi la spécification du comportement à adopter lors de la transition d'un état de QdS à un autre.

Condition d'utilisation des analyses. QuO effectue une analyse dynamique des propriétés de QdS. À partir de la spécification CDL, QuO intègre automatiquement le mécanisme d'observation dans la plate-forme CORBA en utilisant une approche de tissage d'aspect. Lorsque le mécanisme d'observation identifie un changement d'état, il exécute la méthode codant le comportement associé à la transition de l'état. Cette approche nécessite donc la capture des informations extra-fonctionnelles.

Incrémentale. QuO n'est pas une approche incrémentale. Au contraire, lors de chaque modification des propriétés de QdS, il est nécessaire de les recompiler.

Niveau des messages d'erreurs. QuO n'est pas dédié pour la validation dynamique d'un système. C'est une approche qui permet d'exécuter un comportement d'adaptation en réaction à une transition d'état de QdS. En conséquence, pour utiliser QuO en tant qu'outil d'analyse dynamique, les développeurs ont besoin de coder un comportement d'adaptation dont le rôle est d'afficher un message d'erreur explicite.

C) Service WEB

Il existe de nombreux travaux portant sur la spécification de la QdS pour les services WEB, comme par exemple WSLA (WSLA pour *"WEB Service Level Agreement"*) [IBM03] ou WS-agreement [For07]. Le standard WSLA permet de définir un accord de qualité de services entre un fournisseur et un utilisateur de services. Ces spécifications sont écrites en XML, mais le concept général est similaire à CQML. La spécification décrit les propriétés de garantie offertes par le service et les propriétés requises par le service.

Condition d'utilisation des analyses. Il existe différents travaux proposant des méthodes d'analyse statique. Par exemple, ORBWork [CSM+04] est une approche d'analyse statique qui prédit les propriétés de QdS d'une architecture orientée services. Pour réaliser cela, il compose les propriétés de QdS de chaque service en fonction des opérateurs de composition du flot de contrôle, comme la séquence, l'alternative ou le parallélisme. D'autres travaux se focalisent sur l'analyse dynamique. Par exemple, la spécification WSLA explique comment les événements de QdS doivent être observés, c'est-à-dire comment les sondes de QdS doivent être intégrées dans la plate-forme, et comment les contraintes de QdS doivent être dynamiquement évaluées.

Incrémentale. ORBWork propose un mécanisme de vérification incrémentale. Il met à jour les prédictions de QdS de l'architecture en fonction des modifications du flot de contrôle.

Niveau des messages d'erreurs. Les travaux portant sur les analyses statiques fournissent des messages d'erreur de haut niveau. Ces messages indiquent les éléments de conception qui sont la source du problème. Les approches dynamiques produisent généralement des messages de niveau implémentation dans la mesure où ces approches se basent sur le code des services.

D) Bilan

La gestion des propriétés de QdS est très dépendante du modèle utilisé. Très peu de modèles fournissent un support des analyses statiques. Généralement ces approches sont issues du domaine des services WEB. Elles ont besoin de connaître le flot de contrôle de l'application, c'est-à-dire son comportement, ainsi que les propriétés de QdS de chaque composant et service. De plus, dans le cas des analyses incrémentales, la capture des informations concernant la modification du comportement est requise.

D'autre part, la majorité des approches proposent des analyses dynamiques. Ces approches ont besoin de capturer les informations extra-fonctionnelles provenant de la plate-forme. Cette capture repose sur l'intégration de sondes dans la plate-forme qui ont pour rôle d'observer les évolutions des valeurs des propriétés de qualité de services.

3.2.6 Bilan

La table 3.1 contient la comparaison non exhaustive que nous avons effectuée sur les différents travaux portant sur la spécification et l'analyse des propriétés applicatives.

Le premier bilan que nous pouvons tirer de cette comparaison est, qu'à notre connaissance, il n'existe pas de modèle à composants ou de modèle orienté services qui gère les quatre catégories de propriétés applicatives, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. La majorité des modèles, comme CCM [Obj02b] et SCA [BII+07] (cf. lignes 9 et 10) ne supporte pas la spécification des propriétés applicatives. En général, chaque travail se focalise sur la vérification ou l'analyse d'une seule catégorie de propriétés applicatives. L'exemple le plus remarquable est celui du modèle à composants Fractal (cf. ligne 4 du tableau). En effet, la plate-forme d'exécution de référence Julia [BCL+04] ne gère aucune propriété applicative. Chaque catégorie de propriétés applicatives est supportée par des extensions différentes de la plate-forme d'exécution, comme par exemple Fractal-BPC [BABC+09] pour les propriétés comportementales et ConFract [CR99, CMOR07] pour les propriétés de flot de données. Cette multiplication des plates-formes ne permet donc pas à un architecte d'utiliser le modèle Fractal pour vérifier à la fois des propriétés de flot de données et des propriétés comportementales.

Le deuxième point que nous pouvons extraire de cette comparaison concerne le support des analyses incrémentales. Il n'existe que peu de travaux qui se sont intéressés à l'analyse incrémentale des propriétés, néanmoins ce domaine de recherche commence à se développer. Ainsi, des outils portant sur l'analyse incrémentale des propriétés structurelles [Sag06], comportementales [Bar05] et de QdS [CSM+04] ont été développés. Ces travaux améliorent la rapidité de la vérification. Leur utilisation donc est pertinente dans le cadre d'un canevas de développement

Bilan	Structure			Comportement			Flot de données			QdS		
Critère	Catégorie	Incrémental	Messages	Catégorie	Incrémental	Messages	Catégorie	Incrémental	Messages	Catégorie	Incrémental	Messages
UML	statique	oui	conception	statique	non	conception	n/a	n/a	n/a	n/a	n/a	n/a
Fractal	statique (ConFract)	non	oui	statique dynamique (Fractal-BPC)	non	conception	dynamique (ConFract)	non	implémentation	n/a	n/a	n/a
Wright	statique	non	conception	statique	non	conception	n/a	n/a	n/a	n/a	n/a	n/a
Acme/Armani	statique	non	conception	n/a	n/a	n/a	dynamique	non	implémentation	n/a	n/a	n/a
SafArchie	statique	oui	conception	statique	oui	conception	statique	oui	conception	n/a	n/a	n/a
Service Web	n/a	n/a	n/a	statique	non	conception	n/a	n/a	n/a	statique dynamique	oui non	conception implémentation
CCM	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
SCA	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

TAB. 3.1: Support des propriétés applicatives dans les modèles à composants et à services

incrémental et itératif. Donc, afin de permettre l'intégration de ces travaux dans notre canevas, nous avons identifié les informations requises par ces approches : tous ces travaux ont besoin de connaître les modifications qui ont eu lieu dans l'architecture. Nous devrons donc élaborer notre canevas de développement de telle sorte qu'il fournisse ces informations aux outils d'analyse.

Troisièmement, il n'existe aucun travail qui couple les analyses statiques avec les analyses dynamiques. Les travaux sont, soit purement statiques comme Wright [All97], soit purement dynamiques comme la vérification de flot de données dans Armani [JRMWC07]. L'inconvénient majeur de ces approches est un manque de précision dans la prise en compte des interactions partiellement compatibles. Par exemple, les analyses purement statiques peuvent déclarer une architecture fausse car il existe une possibilité de violer une propriété applicative pendant l'exécution, comme le font l'analyse de Wright et de Darwin [MDEK95]. D'autres travaux, comme SafArchie [Bar05] ne déclarent pas l'architecture fausse mais déclenche une alerte. C'est donc ensuite à la charge du développeur de mettre en œuvre les analyses dynamiques en ajoutant des tests dans le code du logiciel pour détecter si le problème survient. De même, les travaux purement dynamiques manquent aussi de précision. Ces travaux insèrent des tests dans l'application sans effectuer d'analyse plus globale, comme dans le cas de Armani [JRMWC07]. Donc, sans analyse statique, l'architecte n'est informé de la violation de la propriété applicative que durant l'étape de validation dynamique, alors qu'il aurait pu l'être dès l'étape de conception. Globalement, les travaux existants n'utilisent pas la connaissance produite par les outils d'analyse statique pour raffinner les analyses dynamiques.

Globalement, notre objectif n'est pas de redéfinir des nouvelles analyses statiques mais de réunir au sein de notre canevas de développement, le maximum de travaux différents portant sur les analyses statiques et dynamiques. En outre, nous avons constater que peu d'approches offrent le support des analyses incrémentales. Cependant, différents travaux dans ce domaine commencent à se développer [CSM+04, Bar05, Sag06]. Dans le cadre d'évolutions d'architectures logicielles, ces travaux sont importants car ils offrent de meilleures performances que les approches non incrémentales. Notre canevas devra donc permettre la réutilisation des approches incrémentales existantes et à venir.

3.3 Gestion du cycle de vie : le projet FAROS

FAROS [4], qui signifie *"composition de contrats pour la Fiabilité d'ARchitecture Orientées Services"* est un projet de recherche ANR (ANR pour *"Agence Nationale de la Recherche"*) de type exploratoire. Ce projet a débuté en décembre 2005 et s'est terminé en octobre 2009. Il est conduit par trois partenaires académiques (l'équipe Triskell de l'IRISA à Rennes, l'équipe Rainbow de l'I3S à Sophia Antipolis et l'équipe Goal du LIFL à Lille) et trois partenaires industriels (EDF R&D, FT R&D et Alicante). Les membres de l'équipe Goal qui ont participé au projet correspondent aux membres de l'équipe-projet INRIA ADAM dont nous faisons partie.

L'objectif du projet est de proposer un canevas permettant la conception de logiciels par composition de services. Le canevas de conception FAROS fournit à l'expert du domaine une vue de conception de haut niveau d'abstraction qui masque les détails liés à la plate-forme d'exécution. Le canevas permet aussi à l'expert du domaine d'exprimer les propriétés applicatives que le logiciel doit respecter. Il assure ensuite la mise en œuvre de la vérification des propriétés applicatives dans la plate-forme. De plus, le canevas FAROS est extensible et multi plates-formes afin de prendre en compte l'hétérogénéité des plates-formes d'exécution.

Le canevas de développement FAROS est un sujet d'étude intéressant dans le cadre de la construction fiable d'architectures logicielles. Il nous a permis de déduire les besoins qui sont requis pour élaborer un canevas dédié pour l'évolution fiable du logiciel, avec notamment la nécessité de conserver un lien causal entre les éléments de conception et les éléments d'exécution. Nous nous sommes donc reposés sur l'expérience acquise tout au long du projet FAROS pour élaborer notre contribution de thèse. La section 3.3.1 présente le principe du procédé FAROS. La

[4] http ://www.lifl.fr/faros

section 3.3.2 détaille la partie clé du procédé qui permet de faire le lien entre la conception de l'architecture et la mise en œuvre de celle-ci dans une plate-forme d'exécution. Finalement, la section 3.3.3 décrit notre retour d'expérience et extrait les concepts clés requis pour élaborer un canevas dédié pour l'évolution du logiciel.

3.3.1 Processus de développement

La solution proposée par FAROS se base sur l'ingénierie dirigée par les modèles (IDM) afin de permettre à l'expert du domaine de concevoir son application avec un haut niveau d'abstraction, puis de projeter automatiquement cette conception sur une plate-forme d'exécution cible. L'approche de FAROS est représentée dans la figure 3.2. Elle repose sur une séparation des préoccupations réalisée au moyen de trois niveaux d'abstraction. Tous les niveaux sont modélisés à l'aide de métamodèles.

Le niveau métier décrit les concepts purement métiers qui sont en relation directe avec une préoccupation donnée, comme par exemple le métier de la diffusion d'informations. Ce niveau contient aussi les contraintes liées à ce métier afin de garantir que la description faite à ce niveau est conforme avec le métier. Chaque métier est spécifié avec un métamodèle métier dédié. De cette manière, l'expert du domaine, non informaticien, peut concevoir son application en manipulant les concepts métiers qu'il connait bien.

Le niveau pivot regroupe les concepts nécessaires à la description d'une architecture à composants ou orientée services indépendamment de la plate-forme cible. Il permet la description de la structure et des diverses propriétés applicatives à l'aide d'un métamodèle pivot unique.

Le niveau plate-forme décrit les structures d'exécution d'une plate-forme d'exécution, comme par exemple Fractal. Chaque plate-forme est modélisée à l'aide d'un métamodèle dédié.

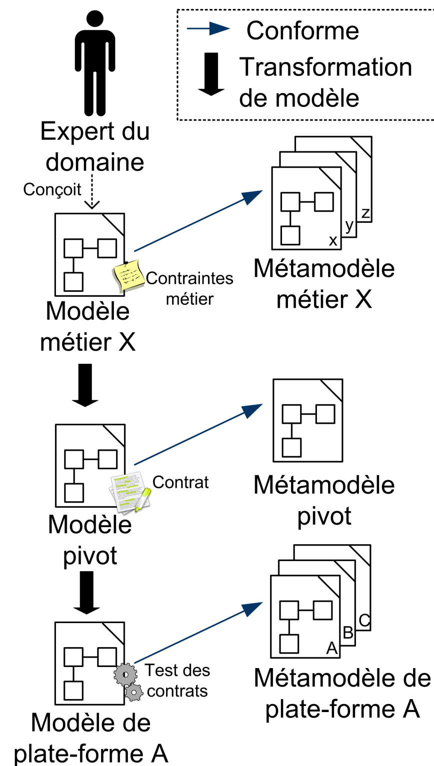

FIG. 3.2: Aperçu de FAROS

Le modèle métier est conçu par l'expert du domaine, tandis que les modèles pivot et plate-forme sont produits, respectivement, par la transformation du modèle métier et du modèle pivot. Les transformations de modèles assurent ainsi, l'intégration dans la plate-forme de la vérification des contrats déclarés au niveau métier.

Dans la suite de cette section, nous nous focalisons sur le métamodèle pivot qui est le coeur du procédé FAROS. Ce métamodèle est un point médian placé entre le niveau métier et le niveau plate-forme. Il permet de factoriser les transformations et donc de faciliter l'extension du canevas FAROS par ajout de nouveaux métamodèles métier et de nouveaux métamodèles de plate-forme.

3.3.2 Métamodèle pivot

Le métamodèle pivot de FAROS est composé de trois parties : la partie structurelle, la partie contractuelle et la partie événementielle.

A) Partie structurelle

La partie structurelle, représentée dans la figure 3.3, a été définie dans le livrable 2.1 [PCW08]. Elle permet de décrire la structure d'une architecture à composants ou orientée services indépendamment de la plate-forme sous jacente. Elle contient les quatre concepts communs aux architectures logicielles présentées dans la section 2.1. Une entité (`Entity`) correspond à un composant ou à un service. Chaque entité possède des ports fournis (`ProvidedPort`) et requis (`RequiredPort`) contenant un ensemble d'opérations (`Operation`). Un connecteur (`Connector`) représente une interaction entre un port requis et un port fourni. Le système correspond à l'assemblage d'entités interconnectées.

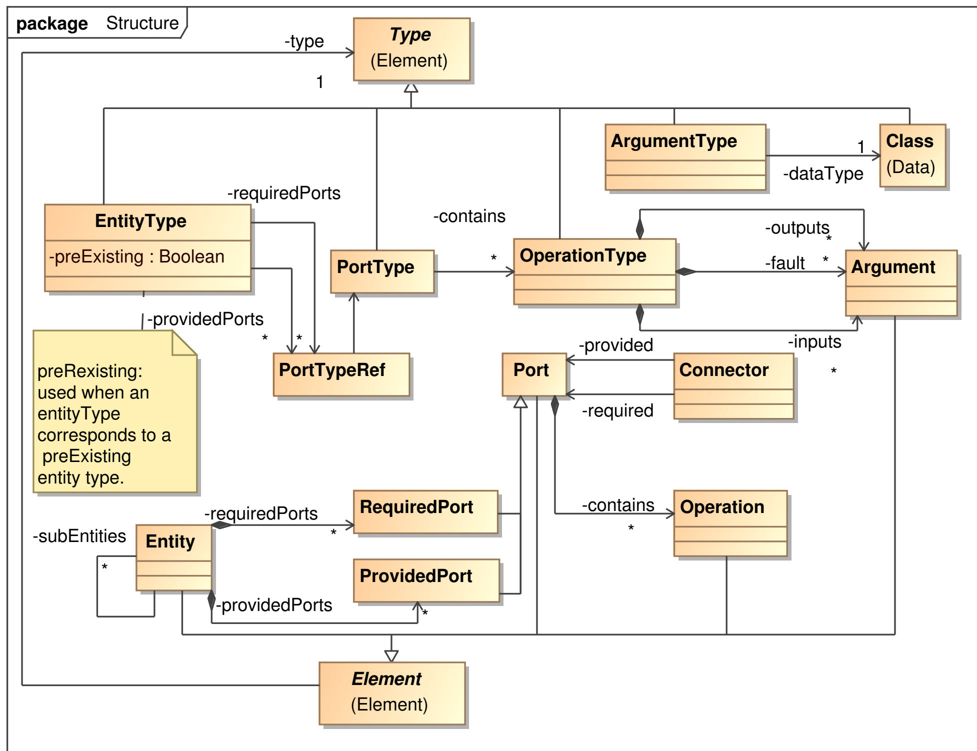

FIG. 3.3: Métamodèle pivot : structure

B) Partie contractuelle

La partie contractuelle du métamodèle pivot est représentée dans la figure 3.4. Cette partie est aussi définie dans le livrable 2.1 [PCW08]. Elle se base sur la spécification de WSLA (WSLA pour *"Web Service Level Agreement"*) [IBM03]. Elle permet de définir un accord sur la qualité de service entre un utilisateur et un fournisseur de service. Un contrat (`Contract`) représente un accord entre plusieurs participants et résulte d'une composition des clauses des participants. Une clause (`Clause`) contient une spécification contractuelle (`Specification`) qui exprime la contrainte requise par le participant. L'agrément (`Agreement`) d'un contrat exprime l'expression de compatibilité entre toutes les clauses du contrat. L'action garantie (`ActionGuarantee`) spécifie l'action à réaliser en cas de violation du contrat.

Ce métamodèle permet aussi de décrire la projection du contrat dans la plate-forme, c'est-à-dire le mécanisme d'analyses dynamiques du contrat. Concrètement cette partie permet de décrire où et quand ont lieu les observations et les analyses dynamiques nécessaires à la mise en œuvre des contrats dans les plates-formes. Concrètement, un événement (`Event`) représente une modification dans la plate-forme d'exécution. Il spécifie le moment où l'analyse dynamique doit être effectuée. Chaque spécification est associée à un checkeur (`Checker`) qui est l'entité responsable de l'analyse dynamique de la spécification lorsqu'un événement se produit. Le rôle

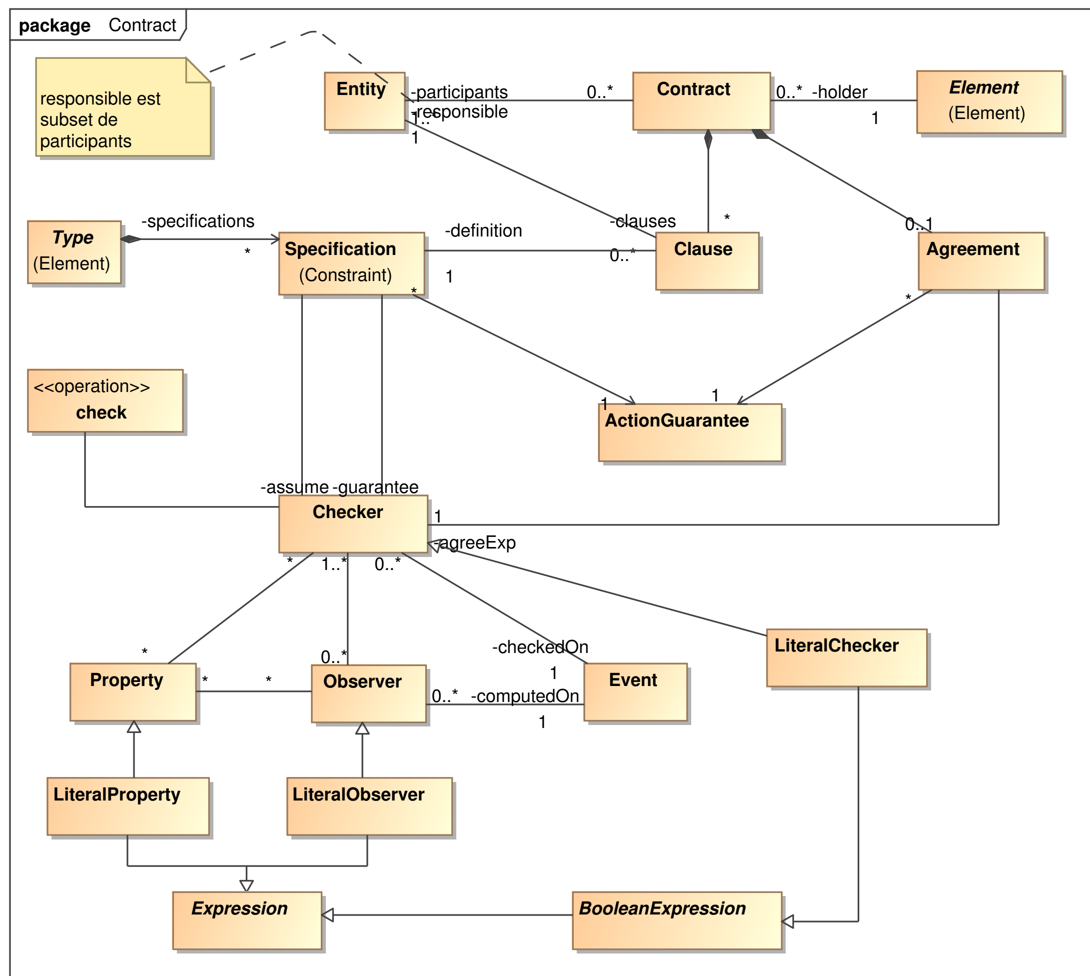

FIG. 3.4: Métamodèle pivot : contrat

d'un observeur (`Observer`) consiste à capturer un événement de la plate-forme et à déclencher l'exécution du checker associé.

C) Partie événementielle

La partie événementielle est représentée dans la figure 3.5. Elle résulte d'une étude comparative des différentes plates-formes d'exécution réalisée dans le livrable 2.3 [DBFC+08]. Ce métamodèle permet de se rapprocher des modèles d'exécution des plates-formes. Il modélise les événements relatifs au cycle de vie et aux interactions des entités. Ce métamodèle contient trois grandes catégories d'événements.

- La première catégorie concerne les événements du cycle de vie, c'est-à-dire la gestion des connexions entre les entités (`AfterAddBinding`, `BeforeAddBinding`, `AfterRetractBinding` et `BeforeRetractBinding`), la gestion du démarrage des entités (`StartEntity`, `StopEntity`, `AfterStartEntity`, `AfterStopEntity`), la gestion du démarrage du système (`StartSystem`, `StopSystem`) et la gestion de l'enregistrement des entités (`RegisterEntity`, `AfterRegisterEntity`, `UnregisterEntity`, `AfterUnregisterEntity`).
- La deuxième catégorie d'événements est composée de deux parties. La première partie correspond aux événements de communication entre les entités (`RequestSending`, `ResponseReceiving`). La seconde partie spécifie les événements de contrôle d'exécution des opérations avec le déclenchement et la fin de l'opération (`OperationEntry`, `OperationExit`).
- Le troisième catégorie d'événements ne contient que l'événement `ApplicationEvent` qui correspond à un événement qui provient de l'application.

3.3.3 Retour d'expérience

L'approche FAROS permet à un expert du domaine de spécifier son logiciel en manipulant exclusivement des concepts métier. De plus l'expert du domaine peut aussi déclarer les contraintes métiers que le logiciel doit satisfaire. Le procédé FAROS se charge alors de projeter la description métier vers une plate-forme à composants ou orientée services, ainsi que de mettre en œuvre les analyses dynamiques des contraintes dans la plate-forme cible. Ce mécanisme de projection repose sur l'ingénierie dirigée par les modèles. Cette approche de développement descendante et linéaire assure un processus de développement uniforme et évite donc l'introduction d'incohérence entre les étapes de conceptions.

En outre, le procédé FAROS est multi plates-formes. Ce support repose sur un modèle pivot, indépendant de toute plate-forme. Ce modèle pivot contient toutes les informations nécessaires pour décrire l'architecture et pour analyser les contraintes, c'est-à-dire qu'il permet d'indiquer le moment et le lieu où il faut effectuer l'analyse dynamique, ainsi que l'analyse à effectuer.

Grâce au retour d'expérience que avons nous acquis lors de notre participation à ce projet, nous avons pu identifier les besoins indispensables pour élaborer un canaves agile pour l'évolution fiable d'architecture logicielle. D'une part, le procédé FAROS n'est pas adapté pour faire évoluer un logiciel car il n'offre pas de cycle de développement itératif et incrémental. En effet, lors de chaque transformation entre les modèles métier, pivot et plate-forme, FAROS ne conserve pas la traçabilité des éléments. En conséquence, lorsqu'une contrainte est violée, les informations d'erreur produites sont spécifiques à la plate-forme utilisée. Donc l'expert du domaine doit analyser le résultat afin de retrouver le concept métier qui est la source de l'erreur, ce qui augmente considérablement la complexité de la validation du logiciel. Le premier besoin est donc de conserver un lien bidirectionnel entre les éléments conceptuels et les éléments d'exécution.

D'autre part, le mécanisme d'analyse dynamique des contraintes de FAROS est très dépendant des fonctionnalités offertes par la plate-forme cible. En effet, FAROS intègre directement les analyses dynamiques des contraintes dans la plate-forme en se reposant sur les fonction-

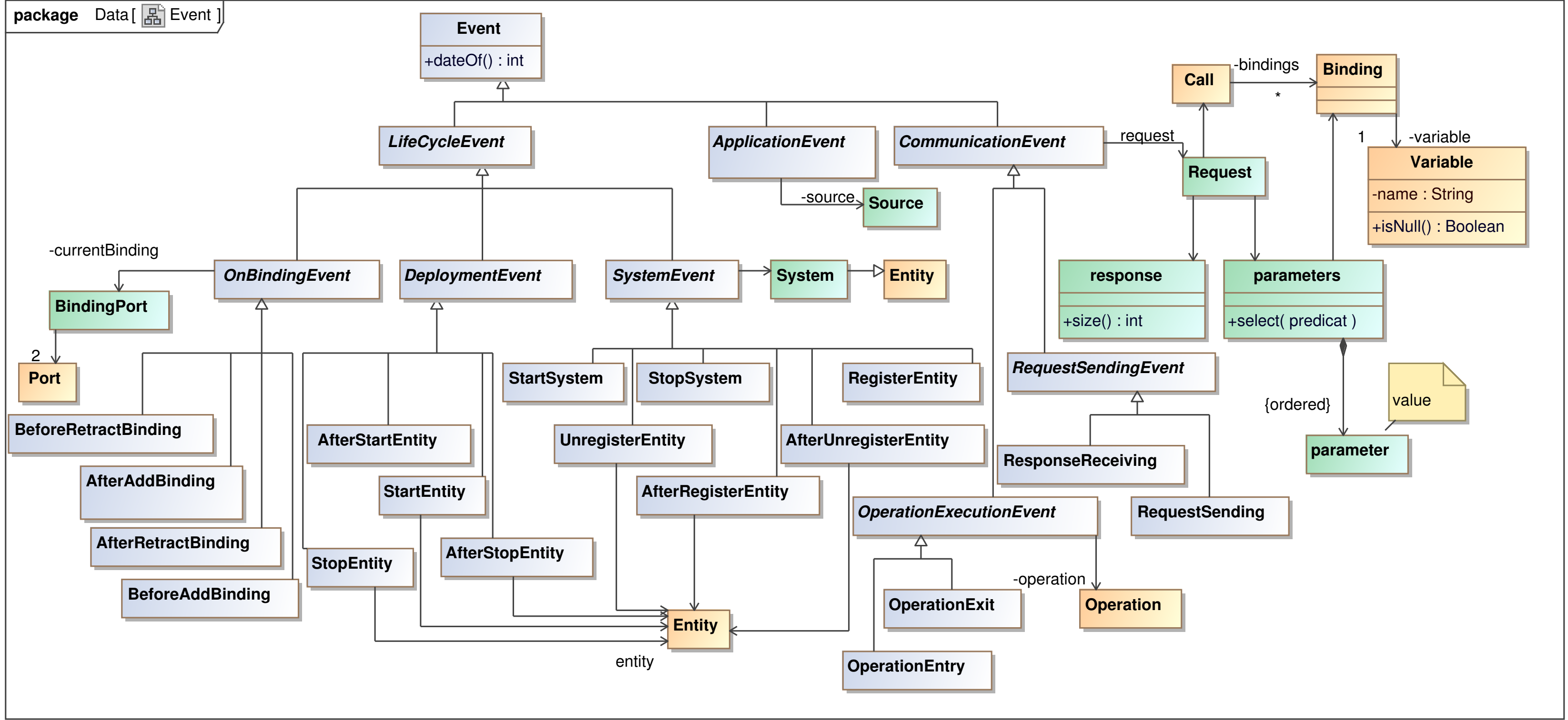

FIG. 3.5: Métamodèle pivot : événement

nalités de validation de contraintes offertes par celle-ci. En conséquence, selon la plate-forme cible choisie, toutes les contraintes métiers ne peuvent pas être vérifiées. Le second besoin est donc d'isoler de manière générique le mécanisme d'analyses dynamiques des contraintes à l'extérieur des plates-formes d'exécution.

Notre objectif sera donc d'exploiter les avantages de FAROS pour le développement uniforme et le couplage des étapes de conception tout en palliant le manque de support lié aux analyses dynamiques dans les plates-formes d'exécution et en se reposant sur un cycle de développement itératif et incrémental.

3.4 Bilan de l'état de l'art

Cette section présente le bilan de l'état de l'art. La section 3.4.1 tire un bilan sur les principaux travaux étudiés dans ce chapitre. Ensuite la section 3.4.2 établit le cahier des charges de notre canevas de développement logiciel et énumère les propriétés attendues par notre approche.

3.4.1 Conclusion

Les systèmes logiciels récents deviennent de plus en plus complexes et ils ont besoin d'être fiables. De plus, leur durée de vie tend à s'allonger. En conséquence ils ont besoin d'évoluer pour s'adapter aux besoins des utilisateurs et aux changements qui se produisent dans le contexte d'exécution.

Ce chapitre de l'état de l'art a donc mis en lumière le panorama des différents travaux qui s'intéressent à l'évolution du logiciel lors des différentes étapes du cycle de développement, c'est-à-dire la conception, l'implémentation, le déploiement et l'exécution. Tous ces travaux permettent d'isoler une nouvelle fonctionnalité dans un élément de conception qui correspond à un patron d'architecture, un aspect ou une règle d'adaptation. Dans un souci de faciliter l'intégration des nouvelles fonctionnalités dans le logiciel, des mécanismes, comme le tissage ou la boucle de contrôle, réalisent cette tâche automatiquement sans altérer les fonctionnalités du logiciel qui ne sont pas concernées par l'évolution. Néanmoins, le support des évolutions reste limité, c'est-à-dire que ces différents travaux ne permettent pas de supprimer ou de modifier des fonctionnalités existantes.

De plus, après une évolution, le logiciel résultant peut potentiellement violer les propriétés applicatives. Or très peu de travaux fiabilisent les adaptations. Ce chapitre a donc listé un éventail d'approches portant sur la validation d'un logiciel à composants ou orienté services. Cette étude comparative a mis en évidence que, bien qu'il existe une multitude de travaux dans ce domaine, aucune plate-forme à composants ou orientée services ne permet la spécification et la validation des quatre catégories de propriétés applicatives, à savoir les propriétés structurelles, comportementales, de flot de données et de QdS. De plus, nous pouvons noter que les plates-formes qui offrent ce support de validation sont minoritaires et que ce support est disparate selon les plates-formes.

Notre objectif n'est pas de redéfinir de nouvelles analyses, mais d'autoriser l'intégration d'un maximum des outils d'analyse existants dans notre canevas de développement agile. Nous avons donc identifié les informations qui sont requises par ces outils d'analyse. Les outils d'analyse statiques ont besoin de connaître la structure et le comportement de l'architecture. Dans le cas des analyses incrémentales, il faut fournir en plus les informations relatives aux modifications architecturales effectuées par l'architecte. Quant aux analyses dynamiques, elles ont besoin d'informations provenant de l'exécution du logiciel. Ces approches récupèrent ces informations en insérant des sondes et des assertions dans le code du logiciel.

3.4.2 Cahier des charges

À partir de notre problématique définie dans l'introduction de ce document, cette section établit le cahier des charges de notre canevas de développement logiciel. Toutes nos décisions

ont été motivées par le fait que notre canevas ne doit en aucun cas être ad-hoc. Au contraire, il doit être générique et extensible.

Cycle de développement. Notre canevas logiciel doit permettre aux différents acteurs (architectes, développeurs, administrateurs et testeurs) d'élaborer un système logiciel fiable et de pouvoir le faire évoluer en fonction des nouveaux besoins du client, ainsi qu'en fonction des changements qui se produisent dans les plates-formes technologiques. De plus, notre solution doit prendre en compte les évolutions complexes, c'est-à-dire supporter la suppression et la modification des fonctionnalités existantes. Pour réaliser cela, les acteurs ont besoin de pouvoir itérer entre les phases de conception et de validation du logiciel de manière uniforme. En conséquence, le canevas de développement logiciel doit se baser sur un cycle de développement itératif et incrémental. De plus, afin d'éviter toute introduction d'incohérence lors du passage entre chaque étape, le canevas doit guider les différents acteurs. Les étapes du cycle de développement doivent donc être très fortement couplées. Ce couplage pourra s'appuyer sur la conservation d'un lien causal bidirectionnel entre les éléments de conception et d'exécution.

Validation Notre canevas doit fiabiliser les évolutions. Dans notre approche, nous supposons que l'infrastructure d'acceuil des composants et des services est fiable. Nous nous focalisons donc sur la fiabilisation de l'application métier. Plus précisément, notre objectif est de garantir que les interactions entre les composants/services sont compatibles. Comme présenté dans la section 3.2, il existe un très grand nombre de travaux qui porte sur la validation des interactions dans une architecture logicielle. Ces travaux sont complémentaires et permettent la validation structurelle, comportementale, de flot de données et de la qualité de services de l'architecture. Notre canevas de développement logiciel devra donc être capable d'exploiter ces différents travaux indépendamment de la plate-forme d'exécution. Pour cela, notre canevas devra permettre la factorisation et la réutilisation des outils d'analyse existants pour chacune des plates-formes d'exécution gérées par notre canevas. En outre, afin de pallier les limitations d'une analyse exclusivement statique, notre canevas devra autoriser le couplage entre les analyses statiques et les analyses dynamiques. Afin d'atteindre cet objectif, notre canevas devra fournir aux outils toutes les informations requises pour effectuer les analyses statiques, dynamiques et incrémentales.

Plate-forme à composants/services. Comme présenté dans le chapitre 2, il existe une grande variété de plates-formes à composants et orientées services. Chaque plate-forme a ses avantages et ses inconvénients. Par exemple, certaines plates-formes sont légères et facilitent donc l'exécution de logiciel dans un environnement restreint, comme des PDA, tandis que d'autre plates-formes sont plus lourdes mais fournissent nativement un ensemble de services techniques, comme des services transactionnels et de courtage. Donc, afin de pouvoir exploiter les différentes plates-formes à composants et orientées services existantes et à venir, notre cadre de développement logiciel devra être générique et extensible afin de permettre la modélisation du système logiciel indépendamment de la plate-forme d'exécution cible. Il devra donc y avoir une séparation claire entre le coté générique du canevas permettant la modélisation du système logiciel et son coté extensible autorisant l'exécution du système sur différentes plates-formes. De manière pragmatique, cette séparation devra faciliter l'intégration du support de nouvelles plates-formes dans le canevas, mais ceci devra être validé par des expérimentations. Notre canevas pourra, pour faire cela, s'appuyer sur les concepts communs aux différents modèles à composants et orientés services qui ont été identifiés dans le chapitre 2 et pourra exploiter le mécanisme de métamodèle pivot défini dans le projet FAROS.

Deuxième partie

Contributions

Chapitre 4

CALICO : un cadre générique pour la conception agile d'application

Sommaire

4.1 Motivation **69**
4.2 Exemple **70**
4.2.1 Scénario d'utilisation 70
4.2.2 Propriétés applicatives du système DMP 71
4.3 Présentation de CALICO **72**
4.3.1 Architecture de CALICO 72
4.3.2 Cycle de développement itératif et incrémental 75
4.3.3 Exemple 76
4.4 Organisation de la seconde partie **77**

Ce chapitre donne un aperçu de notre contribution : le canevas de développement agile CALICO, signifiant *"Component AssembLy Interaction Control FramewOrk"*, pour la conception et l'évolution fiable d'applications. Pour commencer, la section 4.1 présente les motivations qui nous ont conduits à élaborer CALICO. Puis, la section 4.2 expose l'exemple qui va servir à illustrer les concepts importants de CALICO dans la suite de ce document. La section 4.3 décrit l'architecture de CALICO et explique brièvement le fonctionnement de son cycle de développement itératif et incrémental. Finalement la section 4.4 donne l'organisation de la seconde partie de ce document.

4.1 Motivation

Comme énoncé dans le chapitre d'introduction, les logiciels deviennent de plus en plus complexes et leur durée de vie s'allonge. Or, les exigences applicatives et qualitatives des utilisateurs peuvent changer au cours du temps. En conséquence, il est nécessaire de faire évoluer le logiciel afin de l'adapter aux nouveaux besoins des utilisateurs et aux changements de l'environnement d'exécution. Néanmoins, l'évolution du logiciel reste une tâche difficile et source d'erreurs. Des incohérences peuvent être introduites dans le logiciel et le logiciel résultant peut donc ne pas respecter les exigences des utilisateurs. Ainsi, dans un souci de fiabiliser les évolutions, il est nécessaire que chaque évolution soit validée statiquement et dynamiquement. De plus, l'architecte a besoin de corriger et de mettre au point sa conception afin de résoudre les erreurs détectées pendant la validation des évolutions du logiciel.

Notre objectif est donc de créer un canevas de développement agile pour la conception et l'évolution fiable de logiciels à composants ou orientés services. Dans le cahier des charges établi d'après le bilan de l'état de l'art dans la section 3.4.2, nous avons identifié les bonnes propriétés que notre canevas doit posséder. Naturellement, afin de permettre un développement agile, notre canevas a besoin de supporter un développement basé sur un cycle itératif et incrémental. En outre, il est nécessaire que notre canevas guide les différents acteurs (l'architecte, les développeurs et les testeurs) dans leur développement. Concrètement, il doit permettre l'évolution fiable en couplant fortement les différentes étapes du cycle, c'est-à-dire la conception, l'implémentation, le déploiement et la validation statique et dynamique.

D'autre part, dans l'état de l'art nous avons aussi identifié qu'il existe une très grande variété de modèles et de plates-formes à composants ou orientés services. Dans notre approche, nous souhaitons exploiter le potentiel de la majorité de ces modèles et de ces plates-formes.

Notre canevas a donc besoin d'être flexible et extensible pour autoriser l'intégration d'outil d'analyse et de plates-formes d'exécution existants. Pour réaliser cet objectif, nous désirons nous reposer sur le principe de séparation des préoccupations du génie logiciel afin de séparer le coté générique du canevas permettant la modélisation du système logiciel et son coté extensible autorisant l'exécution du système sur différentes plates-formes.

4.2 Exemple

Cette section introduit l'exemple qui sera utilisé dans la suite du document. Elle illustre aussi les différentes exigences qu'un architecte logiciel a besoin d'exprimer. Cet exemple est extrait d'une application plus générale qui est utilisée comme cas d'étude. Cette application correspond au système du Dossier Médical Personnel [Nun] (DMP) qui sert de démonstrateur dans le projet FAROS [1] [Del06]. Elle est plus amplement décrite dans le livrable FAROS du démonstrateur DMP [MWF+09].

Cette section commence par exposer le scénario de l'application, puis elle décrit les propriétés applicatives que l'architecte logiciel désire spécifier afin de pouvoir construire un système DMP plus robuste.

4.2.1 Scénario d'utilisation

Le métier de ce système est de pouvoir délivrer à des professionnels de santé les informations médicales nécessaires pour soigner leurs patients. La figure 4.1 représente un extrait de ce système. Toutes les informations médicales, qui peuvent être par exemple des analyses biologiques, des radiographies ou des ordonnances, sont sauvegardées dans la base de données `MedicalServer`. Un professionnel de santé se sert d'une interface cliente, représentée par l'élément `Client` sur la figure 4.1, pour consulter ces données.

La première préoccupation est liée à l'authentification. En effet, n'importe qui n'est pas autorisée à consulter les données médicales d'un patient. L'architecture de ce système doit donc fournir un mécanisme d'authentification. L'élément architectural `Authentication` permet d'authentifier un professionnel de santé et d'ouvrir sa session. Cet élément retourne un ticket de session via la fonctionnalité `getTicket` de l'élément architectural `SessionServer`. Pour des raisons de sécurité, (*besoin 1*) la fonctionnalité `getTicket` ne peut être utilisée exclusivement que par l'élément `Authentication` afin d'empêcher qu'un utilisateur non authentifié récupère un ticket de session lui permettant de consulter des données médicales confidentielles. De plus, (*besoin 2*) le ticket de session doit obligatoirement être validé par l'élément `SessionServer` avant de récupérer la moindre information médicale afin de détecter par exemple si la session a expiré.

La seconde préoccupation est liée au problème de fiabilité du système. Le système DMP doit être capable de gérer des données médicales très hétérogènes. En effet, ces données peuvent être

[1] http ://www.lifl.fr/faros

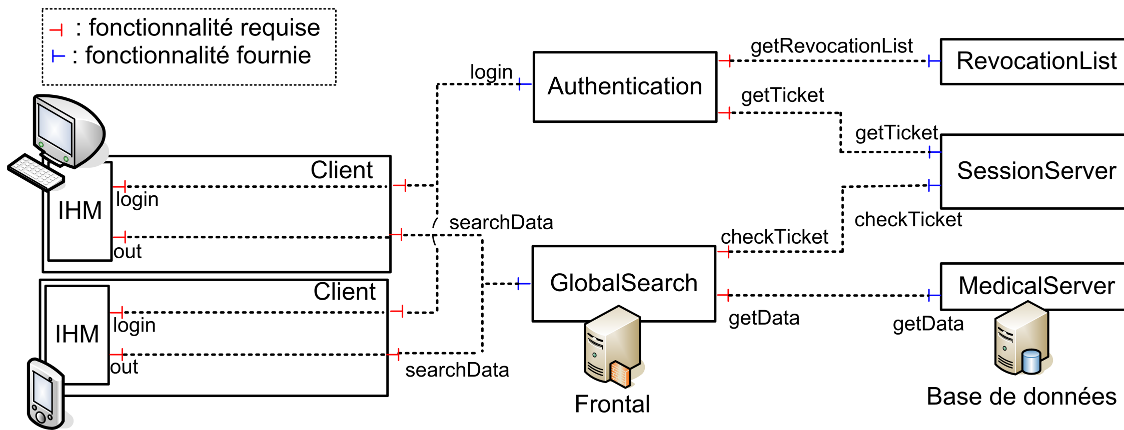

FIG. 4.1: Architecture du système DMP

des ordonnances textuelles de quelques kilooctects et aller jusqu'à des vidéos d'échographies de plusieurs gigaoctets. De plus les terminaux utilisés pour consulter les données médicales, ainsi que le type de connexion internet sont eux aussi hétérogènes. Par exemple, quand le professionnel de santé est en consultation dans son cabinet, il peut se servir de son puissant ordinateur de bureau doté d'une grande capacité mémoire et d'un large écran plat, le tout, connecté à un réseau Ethernet gigabyte. Mais il peut aussi se servir d'un petit PDA ayant évidemment des contraintes matérielles plus restreintes et utilisant une connexion sans fils GPRS lorsque le professionnel de santé est en déplacement chez un patient ou dans une ambulance. Prendre en compte cette hétérogénéité de manière fiable est une préoccupation critique. Le système DMP doit être capable de déterminer à priori si (*besoin 3*) une information médicale peut être affichée correctement sans perte de données et (*besoin 4*) dans un délai convenable sur un terminal en fonction des caractéristiques de l'information et du terminal. En conséquence, l'architecte a besoin de prendre en compte le profil des terminaux et les types des informations échangées pendant l'étape de conception du système afin de pouvoir élaborer un système DMP fiable.

4.2.2 Propriétés applicatives du système DMP

Dans le but de concevoir un système DMP robuste, l'architecte logiciel a besoin d'exprimer les exigences qualitatives énumérées dans la section précédente. Concrètement, il a besoin de spécifier plusieurs propriétés applicatives, dont les propriétés structurelles, comportementales, de flot de données et de QdS suivantes.

Pour des raisons de sécurité, (*besoin 1*) l'architecte logiciel a besoin de pouvoir ajouter une propriété structurelle `S1` sur la fonctionnalité `getTicket` de l'élément architectural `SessionServer` dans le but de limiter exclusivement l'utilisation de cette fonctionnalité à l'élément `Authentication` (cf. figure 4.1).

Pour des raisons d'authentification, (*besoin 2*) l'architecte doit être capable d'exprimer que le ticket de session doit être validé à l'aide de la fonctionnalité `checkTicket` de l'élément `SessionServer` avant d'appeler la fonctionnalité `getData` de l'élément `MedicalServer`. Pour exprimer cela, il a besoin d'ajouter une propriété comportementale `Bi` sur chaque élément architectural du système DMP afin de décrire le flot de contrôle de ces éléments. Ces propriétés pourront ensuite être utilisées pour garantir que le système DMP ne possède aucun interblocage.

Pour des raisons de fiabilité, (*besoin 3*) l'architecte logiciel veut avoir la garantie qu'en aucun cas un terminal ne reçoive des données médicales qui violent le profil du terminal, entraînant un disfonctionnement du système DMP. En effet, par exemple un dépassement mémoire sur un PDA va causer un arrêt brutal du système DMP, ce qui n'est pas envisageable dans un cas d'urgence médicale. Donc, l'architecte a besoin d'exprimer des propriétés de flot de données. Par exemple, il peut avoir besoin de spécifier, via une propriété `D1`, que les données médicales reçues

sur le terminal de type PDA Nokia N800 doivent avoir une taille inférieure à 10 mégaoctets. Il peut aussi avoir besoin de spécifier que ce terminal ne peut lire que des documents de type texte ASCII et des images JPG. Dans la mesure où définir le profil d'un terminal n'est pas du tout trivial, il est impossible de partir du postulat que l'architecte va spécifier des propriétés correctes dès la première version de sa conception. Au contraire, l'architecte a besoin de valider dynamiquement dans l'environnement réel, c'est-à-dire directement sur le PDA cible, le comportement du système DMP. L'exécution des scénarios d'utilisation sur une machine virtuelle simulant le PDA peut aussi être envisagée. Mais, de part notre expérience, selon les simulateurs utilisés, les résultats produit sont moins précis qu'avec une exécution réelle. Par exemple, la machine virtuelle d'Android [2] ne prend pas en compte la simulation du niveau de la batterie du PDA ou des déconnexions réseaux. Dans la pratique, l'architecte doit itérer plusieurs fois entre la conception et la validation dynamique afin de pouvoir raffiner la spécification du profil des terminaux en fonction du comportement réel du système en exécution.

Pour des raisons de performance, (*besoin 4*) l'architecte doit pouvoir exprimer les contraintes de délai d'affichage des données sur le terminal. En effet, le professionnel de santé ne peut pas attendre indéfiniment l'affichage d'une information médicale. De la même manière que précédemment, l'architecte a besoin de spécifier une propriété de QdS `QdS1`, mais il ne peut pas fixer arbitrairement la valeur du délai d'affichage. L'architecte doit prendre en compte les exigences de chaque professionnel de santé. Pour réaliser cet objectif, il doit aussi itérer entre l'étape de conception pour ajuster les propriétés de QdS, et l'étape d'exécution permettant aux professionnels de santé d'exprimer leurs exigences.

Dans la suite de ce document, nous illustrons avec ce scénario les avantages que CALICO offre à l'architecte pour concevoir et valider statiquement et dynamiquement un système logiciel.

4.3 Présentation de CALICO

Cette section présente notre contribution CALICO qui signifie *"Component AssembLy Interaction Control framewOrk"*. CALICO est un canevas de développement logiciel agile pour les systèmes à composants et à services (cf. figure 4.2). Il permet à l'ensemble des acteurs (architecte, développeurs et testeurs) de concevoir et faire évoluer le logiciel de manière fiable. Dans ce but, CALICO associe une analyse statique des évolutions avant leur propagation dans le logiciel avec une analyse dynamique des évolutions en exécutant le logiciel dans la plate-forme. L'architecte logiciel peut donc élaborer et valider son système logiciel à composants ou orienté services en itérant entre les étapes de conception et les étapes de validation. Dans la première section, nous donnons un aperçu global de l'architecture de CALICO. Nous expliquons aussi l'impact que les objectifs ont eu sur les choix de conception de notre canevas. Ensuite dans la seconde section, nous détaillons les étapes du cycle de développement proposées par CALICO.

4.3.1 Architecture de CALICO

Afin de répondre à la problématique liée à la séparation des préoccupations entre le côté générique dédié à la conception et le côté extensible dédié à la validation dynamique sur les différentes plates-formes, nous avons décidé d'adopter une démarche d'ingénierie dirigée par les modèles. Nous avons donc choisi d'utiliser l'approche *"Model at runtime"*. Avec cette approche, les modèles sont gardés dans le canevas CALICO, pendant l'exécution du logiciel. En conséquence, nous avons organisé CALICO en deux niveaux : un niveau modèle et un niveau plate-forme qui sont mis en évidence sur la figure 4.2. Le niveau modèle de CALICO permet à l'architecte logiciel de concevoir et de faire évoluer son logiciel. Il contient les modèles du système logiciel en cours d'exécution. Le niveau plate-forme contient le système en cours d'exécution et permet la validation et la mise au point dynamique du logiciel. Les deux niveaux sont gardés

[2] http ://www.android.com

synchronisés de telle sorte que l'architecte dispose toujours, au niveau modèle, d'une vision actualisée de son architecture qui est conforme avec le logiciel qui s'exécute.

Le niveau modèle. Pour concevoir et faire évoluer son logiciel, l'architecte a besoin d'assembler des composants et des services, ainsi que de spécifier les diverses propriétés applicatives de son logiciel. Dans cet objectif, CALICO offre à l'architecte un ensemble de métamodèles de description d'architecture. Ces métamodèles sont indépendants de tout modèle à composants ou orienté services. La spécification complète d'un système logiciel est exprimée à l'aide de cinq métamodèles (cf. figure 4.2). Le métamodèle de *structure du système* permet la description des composants ou des services ainsi que leur assemblage. Le métamodèle des *contrats structurels* sert à spécifier les propriétés applicatives structurelles de l'assemblage de composants ou de services. Le métamodèle des *contrats comportementaux* est utilisé pour décrire le flot de contrôle entrant et sortant des composants ou des services. Le métamodèle des *contrats de flot de données* permet de définir des hypothèses et des garanties sur les valeurs des messages échangés entre les composants ou les services. Le métamodèle des *contrats de qualité de services* (QdS) sert à spécifier des propriétés extra-fonctionnelles de l'assemblage, comme par exemple des propriétés de sécurité ou liées aux performances.

Afin de s'assurer que la description de l'architecture est correcte et qu'aucune propriété applicative n'est violée, la cohérence de l'architecture est vérifiée statiquement par l'outil d'analyse des interactions entre les composants et les services. L'analyse détermine toutes les interactions entre les composants et les services à l'aide du métamodèle de la structure du système. Pour chaque interaction, elle analyse les propriétés applicatives des composants et des services, qui participent à l'interaction, qui sont spécifiées dans les quatre métamodèles de contrats et en déduit la catégorie de l'interaction. CALICO prend en compte les trois catégories d'interaction (cf. partie B de la section 2.2.5). Une interaction est *compatible* quand toutes les propriétés applicatives sont respectées statiquement. Une interaction est *incompatible* quand une des propriétés, d'un des composants ou d'un des services participant à l'interaction, est violée. Une interaction est *partiellement compatible* quand l'analyse statique ne déclenche aucune erreur mais ne peut pas être complétée car certaines propriétés dépendent de valeurs qui ne peuvent être connues que lors de l'exécution.

Dans le but d'obtenir une architecture cohérente, pour chaque interaction incompatible, l'outil notifie le problème à l'architecte afin que celui-ci corrige la conception de son architecture. Pour chaque interaction partiellement compatible, l'outil dispose de deux métamodèles, le métamodèle de *débogage* et le métamodèle d'*aspect*, qui permettent de définir le lien avec l'étape de validation dynamique, de manière générique. Le métamodèle de *débogage* contient la description de l'ensemble des analyses dynamiques qui doivent être effectuées pendant l'exécution du système. Le métamodèle d'*aspect* repose sur le principe de programmation orientée aspect [KLM+97] et il sert à spécifier des fonctionnalités transverses requises pour observer les données d'exécution nécessaires aux analyses dynamiques. De cette manière, l'outil d'analyse statique peut insérer dans le modèle de débogage les analyses dynamiques qui doivent être effectuées pendant l'exécution du logiciel afin de finaliser la validation des interactions partiellement compatibles.

Le niveau plate-forme. Il permet de valider dynamiquement le logiciel sur une plate-forme d'exécution cible et terminer l'analyse des interactions partiellement compatible. Pour réaliser cet objectif, ce niveau contient le logiciel en cours d'exécution correspondant à l'architecture décrite au niveau modèle. Le logiciel est toujours gardé synchronisé avec la description faite au niveau modèle. Concrètement, CALICO dispose d'un ensemble d'outils de support d'exécution chargé de faire le lien entre le niveau modèle et le niveau plate-forme. Le rôle de ces outils est de coupler fortement les différentes étapes de développement, c'est-à-dire la conception, l'implémentation, le déploiement et l'exécution. Ainsi, ces outils permettent de générer le squelette de code des composants et des services de telle sorte que les développeurs aient seulement besoin d'écrire le code métier. Ensuite, ils instancient l'application sur une plate-forme à composants ou orientée services cible à partir des spécifications au niveau modèle. Ils mettent aussi en

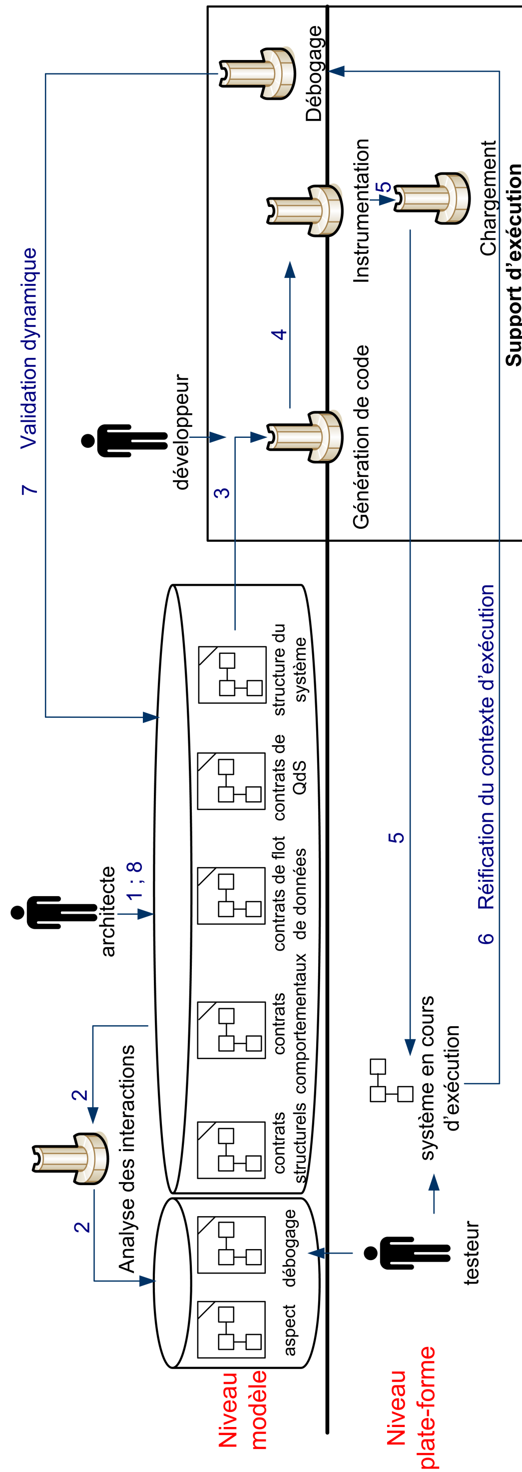

FIG. 4.2: Aperçu de CALICO

œuvre le mécanisme pour réifier au niveau modèle les données d'exécution nécessaires pour la validation dynamique du système.

4.3.2 Cycle de développement itératif et incrémental

CALICO permet aux différents acteurs d'élaborer et de faire évoluer un système logiciel à composants ou orienté services en effectuant un processus de conception itératif et incrémental. À chaque itération, CALICO permet d'alterner entre les activités de conception et les activités de validation dynamique. Les activités de conception correspondent, d'une part, à la définition de la structure et des propriétés du système par l'architecte logiciel. D'autre part, elles consistent à mettre à jour le système afin de corriger les erreurs identifiées durant les activités de validation dynamique et à faire évoluer le système en fonction du retour des informations remontées par les utilisateurs ou la plate-forme, comme par exemple le temps de réponse. Les activités de validation dynamique consistent à l'exécution des scénarios d'utilisation par les testeurs afin de vérifier dynamiquement la compatibilité des interactions entre les composants et les services. Plus précisément, le cycle de développement itératif et incrémental est composé des huit étapes suivantes (voir la figure 4.2) :

1. **Conception** : L'architecte produit une première conception de son système à l'aide des métamodèles de structure du système et de contrats de CALICO. Ainsi, il conçoit l'architecture de son système et spécifie les diverses propriétés structurelles, comportementales, de flot de données et de QdS que le logiciel doit satisfaire.
2. **Analyse statique** : L'*outil d'analyse des interactions* vérifie la cohérence de l'architecture du système, c'est-à-dire qu'il identifie si l'architecture décrite respecte les propriétés applicatives spécifiées. Pour chaque interaction partiellement compatible, il ajoute dans le modèle de débogage l'analyse dynamique qui doit être faite pendant l'exécution du système. Pour chaque interaction incompatible, l'outil notifie l'architecte afin que ce dernier corrige sa conception pour corriger le problème. Tant qu'il reste des interactions incompatibles, les étapes suivantes du cycle ne peuvent pas effectuées. Cela garantit la production d'un système cohérent, c'est-à-dire qui ne viole pas les propriétés applicatives. De plus, cet outil vérifie aussi les contraintes de la plate-forme cible afin de garantir que l'architecture pourra être déployée sur la plate-forme.
3. **Implémentation** : Pour chaque composant ou service qui n'existe pas, l'*outil de génération de code* produit le squelette de code. Lors de cette étape, les développeurs implémentent le code métier des nouveaux composants et services.
4. **Instrumentation** : Cette étape a pour but de faire le lien entre les analyses statiques et dynamiques. Pour réaliser cet objectif, l'*outil d'instrumentation* analyse le modèle de débogage afin d'identifier les analyses dynamiques qui doivent être effectuées. Pour chaque analyse dynamique, l'outil d'instrumentation détermine les données d'exécution nécessaires. Ensuite, l'outil ajoute le support requis pour observer ces données d'exécution. Pour réaliser cet objectif, l'outil insère les mécanismes d'observation dans le modèle d'aspects, instrumente le code de l'application complété par les développeurs et génère des sondes d'observation.
5. **Déploiement** : L'*outil de chargement* de l'application instancie le système sur la plate-forme cible conformément à la description de l'architecture faite au niveau modèle. Concrètement, l'outil génère un modèle contenant la succession des opérations élémentaires à effectuer pour instancier chaque élément nécessaire pour former l'application, comme par exemple ajouter ou supprimer un composant. Ensuite, l'outil interprète ce modèle en appelant l'API spécifique à la plate-forme d'exécution pour instancier le système.
6. **Exécution** : Pendant cette étape, les testeurs exécutent les scénarios d'utilisation du système logiciel, définis pendant l'étape d'analyse des besoins, afin de valider dynamiquement les interactions entre les composants et les services. Ainsi, le système instrumenté réifie les données d'exécution, comme par exemple les changements du contexte d'exécution ou les

valeurs des messages échangés, au niveau modèle sous forme d'évènements. Les testeurs peuvent aussi enrichir les modèles de débogage et d'aspect afin de spécifier les données d'exécutions supplémentaires qu'ils désirent observer. Afin de mettre en œuvre automatiquement le mécanisme d'observation des nouvelles données dans la plate-forme, les étapes 4 et 5 sont redéclenchées.

7. **Validation dynamique** : Au niveau modèle, l'*outil de débogage* analyse les évènements et si besoin, il déclenche l'analyse dynamique appropriée contenue dans le modèle de débogage. Si l'outil identifie une erreur d'exécution, il notifie l'architecte pour que celui-ci corrige sa conception.
8. **Evolution de la conception** : En accord avec les résultats de la validation dynamique, l'architecte peut modifier sa conception pour corriger le problème. L'architecte peut aussi faire évoluer sa conception afin de l'adapter aux nouveaux des besoins des utilisateurs et au changement dans le contexte d'exécution. L'architecte réitère le cycle de développement à partir de l'étape 2 afin d'analyser si les changements dans la conception n'ont pas introduit de nouvelles incohérences.

À la fin du développement, lorsque le système devient stable, CALICO permet à l'architecte la mise en production de son système. Ainsi, CALICO génère les fichiers de description requis par la plate-forme cible afin d'exécuter le système en dehors de CALICO.

Dans son ensemble, CALICO est un canevas dédié pour faire évoluer le logiciel. En effet, lors de chaque itération, le système en exécution n'est pas réinstancié entièrement depuis l'état initial. Au contraire, seuls les changements de conception sont propagés dans le système en exécution grâce à l'exécution d'une adaptation dynamique contrôlée par l'outil de chargement. De plus l'architecte peut aussi modifier sa conception à n'importe quel moment afin de faire face à une évolution inattendue du contexte d'exécution et des besoins applicatifs des utilisateurs.

4.3.3 Exemple

Cette section donne un scénario d'utilisation de CALICO pour concevoir le système DMP décrit précédemment.

(1) Pour commencer, l'architecte conçoit l'architecture du système DMP et ajoute les diverses propriétés applicatives conformément aux spécifications présentées dans la section 4.2.

(2) À partir de cette description, l'outil d'analyse des interactions de CALICO vérifie la cohérence de l'architecture. Il identifie que des interactions qui mettent en jeu les propriétés de flot de données `D1` et de QdS `QdS1` sont partiellement compatibles. Donc, afin de permettre une validation dynamique de ces interactions, l'outil ajoute des analyses dynamiques dans le modèle de débogage. Le rôle de ces analyses est de vérifier que la taille et le type des données médicales qui entrent dans le PDA ne violent pas le profil du PDA, c'est-à-dire que la taille des données est inférieure à 10 mégaoctets et que les données sont des fichiers ASCII ou des images JPG. De plus, le rôle de ces analyses est aussi de tester que le temps de réponse requis pour recevoir une donnée est bien inférieur à 10s. Dans la suite de l'exemple, nous nous focalisons sur les propriétés de flot de données.

(3) L'outil de génération de code produit le squelette de code des composants. Lors de cette étape, les développeurs implémentent le code métier de chaque composant du système.

(4) À partir des informations contenues dans le modèle de débogage, l'outil d'instrumentation instrumente le code des composants afin de capturer la taille et le type des données médicales qui entrent dans le PDA. L'outil génère aussi des sondes pour calculer le délai de réception des données.

(5) L'outil de chargement déploie les composants instrumentés dans la plate-forme cible.

(6) Pendant cette étape, les testeurs exécutent les scénarios d'utilisation du DMP. Par exemple, ils simulent l'utilisation du DMP par un pharmacien et un radiologue. Les pharmaciens n'utilisent le système DMP que pour consulter des ordonnances textuelles et les radiologues peuvent, quant à eux, de plus visionner des radiographies. Pendant l'exécution des scénarios, les données d'exécution sont propagées au niveau modèle.

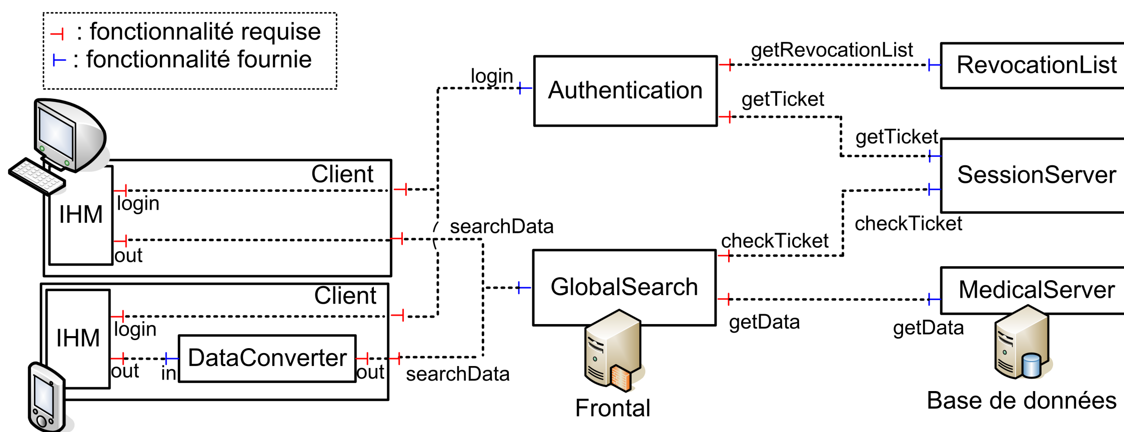

FIG. 4.3: Architecture du système DMP après une évolution

(7) L'outil de débogage utilise les données d'exécution afin de valider dynamiquement les interactions. Dans le cas du pharmacien, aucune erreur dynamique ne survient dans la mesure où le pharmacien ne consulte que des données peu volumineuses, c'est-à-dire des ordonnances textuelles. Dans le cas du radiologue, aucune erreur n'est produite quand il consulte des comptes rendus textuels des radiographies sur le PDA. Néanmoins, l'outil détecte qu'il ne peut pas visionner les radiographies sur son PDA car elles sont trop volumineuses.

(8) L'architecte est informé que la propriété de flot de donnée du PDA est violée quand un radiologue visionne une radiographie sur son PDA. Afin de corriger le problème, l'architecte décide de faire évoluer le système DMP en insérant un nouveau composant `DataConverter` entre le client PDA et le composant `GlobalSearch` dont le rôle est de réduire la taille de la radiographie (cf. figure 4.3). Afin que le gros volume de données ne soit pas transféré au PDA, le composant `DataConverter` est herbergé, avec le composant `GlobalSearch`, sur la machine `Frontal`. L'architecte réitère le cycle de développement : l'outil d'analyse des interactions ne détecte aucune incohérence. L'outil de génération de code produit le squelette de code du composant `DataConverter` qui est complété par les développeurs. L'outil d'instrumentation instrumente le code du composant `DataConverter`. L'outil de chargement modifie dynamiquement le système DMP pour y ajouter le composant `DataConverter`. Pour finir, les testeurs reexécutent les scénarios d'utilisation. Avec cette nouvelle évolution, les radiologues sont autorisés à visionner des radiographies sur le PDA.

4.4 Organisation de la seconde partie

Dans la seconde partie, nous allons détailler comment un architecte élabore un système logiciel de manière itérative et incrémentale en utilisant CALICO. Concrètement, nous allons suivre le cheminement du cycle de développement proposé par CALICO, comme le montre la figure 4.4. Pour commencer, le chapitre 5 présente l'étape conception du système. Il explique les différents métamodèles de CALICO qui permettent à l'architecte de spécifier l'architecture de son application et ses propriétés applicatives. Dans ce chapitre, nous argumentons chacun de nos choix de modélisation.

Ensuite le chapitre 6 décrit le mécanisme d'analyse statique de CALICO. Il détaille aussi le fonctionnement du couplage entre les analyses statiques et les analyses dynamiques.

Puis le chapitre 7 explique le mécanisme de synchronisation entre la conception et l'exécution du système permettant sa validation dynamique. Il présente chacun des outils du support d'exécution de CALICO.

Pour finir le chapitre 8 concrétise nos travaux autour de CALICO. Il commence par présenter l'implémentation de CALICO, puis il expose les expérimentations que nous avons effectuées

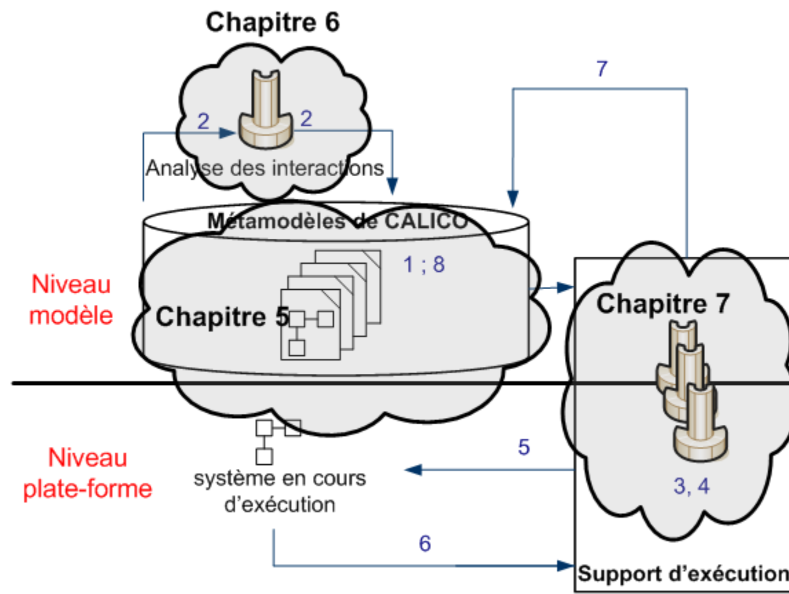

FIG. 4.4: Plan de la suite du document

pour évaluer CALICO.

Chapitre 5

Étape de conception du système

Sommaire

5.1 Motivation 79
5.1.1 Spécification de la structure du système 80
5.1.2 Spécification des propriétés applicatives 80
5.2 Spécification de la structure du système 81
5.2.1 Métamodèle de structure du système 81
5.2.2 Gestion des spécificités des plates-formes 84
5.2.3 Exemple 87
5.2.4 Discussion 89
5.3 Spécification des propriétés applicatives 90
5.3.1 Paradigme choisi pour la spécification 90
5.3.2 Spécification structurelle 91
5.3.3 Spécification comportementale 92
5.3.4 Spécification du flot de données 94
5.3.5 Spécification de qualité de service 97
5.3.6 Bilan 98
5.4 Conclusion 98

Ce chapitre décrit la première étape du cycle de développement contrôlé par CALICO, présentée dans le chapitre 4, c'est-à-dire l'étape de conception de l'architecture. Pour commencer, la section 5.1 détaille les motivations et les objectifs de cette étape. Les sections suivantes décrivent notre contribution. La section 5.2 explique comment un architecte logiciel utilise les différents métamodèles de CALICO pour définir l'architecture de son système, indépendamment de toute plate-forme sous-jacente. Puis la section 5.3 décrit comment l'architecte se sert des métamodèles de CALICO pour spécifier les diverses propriétés applicatives de son système. Durant tout ce chapitre, les éléments importants de CALICO sont illustrés avec des extraits de l'exemple d'application DMP présentée dans le chapitre 4 et décrite dans le livrable FAROS 4-6 [MWF+09].

5.1 Motivation

La première étape du développement correspond à l'étape de conception du système logiciel. Pendant cette étape, l'architecte logiciel a besoin, d'une part, d'assembler les composants et les services qui forment le logiciel, et d'autre part, il a besoin d'exprimer les propriétés applicatives. Nous rappelons donc dans cette section les motivations qui nous ont guidés pour élaborer l'étape

de conception contrôlée par CALICO. Notre première motivation concerne la spécification de la structure du système et notre deuxième motivation porte sur la spécification des propriétés applicatives. CALICO devra remplir les conditions du cahier des charges qui a été établi dans le bilan de l'état de l'art dans la section 3.4.2.

5.1.1 Spécification de la structure du système

La spécification d'une architecture logicielle est devenue une activité centrale et cruciale pour la conception de système logiciel complexe. L'architecte logiciel doit prendre soin de décrire les entités, qui sont des composants ou des services, et d'exprimer la composition de ces entités afin de former le système logiciel désiré. Actuellement, les systèmes logiciels à composants ou orientés services sont décrits avec un langage de description d'architecture (ADL) [MT00] fortement couplé avec la plate-forme d'exécution (cf. section 2.2.3 de l'état de l'art). Ce couplage fort est source de deux inconvénients majeurs que nous allons énumérer ci-dessous.

Premièrement, en raison de ce fort couplage, l'ADL contient des concepts très dépendants de la plate-forme d'exécution et qui sont généralement liés au mode de fonctionnement de la plate-forme. Il y a donc un mélange des préoccupations. L'architecte doit être à la fois un expert métier et un expert de la plate-forme. Or, le rôle de l'architecte est de concevoir le métier de l'application, il n'est donc pas envisageable que l'architecte soit un spécialiste de toutes les plates-formes existantes.

Le deuxième problème est lié au choix de la plate-forme cible. Comme l'ADL est fortement couplé à une plate-forme, le choix d'un ADL pour la conception de l'application va automatiquement contraindre l'utilisation de la plate-forme d'exécution liée à cet ADL. Cette approche va à l'encontre de la séparation des préoccupations entre les étapes de conception et d'implémentation. En effet, c'est le choix de l'architecte, pour un ADL donné, qui va obliger l'utilisation de la plate-forme associée qui sera utilisée pour exécuter l'application, ce qui est avant tout un problème d'implémentation. De plus, si le choix de la plate-forme ne correspond pas aux attentes, alors la migration vers une autre plate-forme cible impose la réécriture complète de la description de l'architecture en utilisant l'ADL de cette plate-forme.

Notre premier objectif est donc de traiter le problème du couplage fort entre l'ADL et la plate-forme d'exécution. La première partie de notre travail a donc consisté en la définition d'une représentation générique permettant de décrire des assemblages de composants ou de services avec une représentation de haut niveau, indépendamment de toute plate-forme d'exécution. De plus, nous ne voulons pas imposer une forme de représentation visuelle. Au contraire, nous voulons permettre à l'architecte de choisir une représentation visuelle de la description de l'architecture qu'il connaît bien, comme une représentation textuelle ou graphique.

5.1.2 Spécification des propriétés applicatives

Comme nous avons pu le voir dans la section 3.2 de l'état de l'art, il existe de nombreuses catégories de propriétés applicatives. Les propriétés typiques portent sur les interactions structurelles et comportementales entre les composants et les services [Hoa85, Mag99]. Ces propriétés sont importantes car elles permettent à l'architecte de spécifier et de comprendre les dépendances entres les différents composants et services formant le système. L'architecte peut aussi avoir besoin de raisonner sur les propriétés de qualité de services du système, comme la performance du système ou la sécurité [FK98, PLS+00]. Finalement, d'autres catégories de propriétés du système peuvent concerner la valeur des messages échangés entre les composants ou les services, comme nous l'avons expliqué avec le scénario DMP. Généralement, ces propriétés sont utilisées pendant l'exécution du système [CR99].

Afin de permettre à l'architecte de spécifier les propriétés applicatives de son architecture, les ADLs sont souvent associés avec des langages de description des propriétés. Cependant, il existe de nombreux ADLs différents. Chaque ADL a été élaboré pour supporter la description d'un ensemble de catégories spécifiques de propriétés. En conséquence, bien qu'il existe une très

large variété de travaux concernant la spécification de propriétés applicatives, ces travaux sont fortement couplés avec un modèle à composants ou orienté services. Ce couplage limite donc leurs utilisations dans le cadre d'autres modèles à composants ou orientés services. L'architecte pourrait réduire ce problème en décrivant la même architecture en utilisant les différents ADLs fournissant le support des propriétés applicatives désirées. Cependant, cette solution entraîne un surcoût lié à la redondance de la description et à la synchronisation des différentes descriptions. Bien que des ADLs génériques comme Acme [GMW00] ou xADL [DdHT01] existent, ils ne supportent principalement que l'expression des propriétés structurelles du système [Mon01]. A notre connaissance, il n'existe aucun ADL qui supporte les propriétés structurelles, comportementales, de flot de données et de QdS.

Notre deuxième objectif est donc de traiter le problème de la spécification des propriétés applicatives. Notre canevas de développement doit permettre d'exploiter la large variété des travaux existant sur la spécification des propriétés applicatives. Concrètement, il doit être extensible afin d'autoriser la réutilisation d'un maximum des travaux existants autour de la spécification des propriétés applicatives, comme par exemple les propriétés structurelles, comportementales, de flots de données et de QdS, présentées dans la section 3.2.

5.2 Spécification de la structure du système

La première étape du cycle de développement est la conception du système. La conception du système inclut la spécification de la structure du système, c'est-à-dire la définition des composants ou des services et leur assemblage. Notre premier objectif a donc été de définir une spécification qui contient tous les éléments nécessaires pour décrire une architecture logicielle avec un niveau de détail suffisant pour modéliser et raisonner sur les interactions entre les composants et les services.

Pour réaliser cet objectif, nous avons choisi de nous reposer sur l'ingénierie dirigée par les modèles. En effet l'IDM a pour but de simplifier la description d'un système sans se soucier des détails et de raisonner sur une abstraction de la réalité. Nous avons donc défini le métamodèle de *structure du système* qui permet à l'architecte de décrire la structure de son architecture indépendamment de la plate-forme d'exécution utilisée. De plus, ce paradigme n'impose pas un style de représentation. Il est possible d'ajouter une surcouche de manipulation visuelle des modèles, comme une surcouche graphique ou textuelle. De cette manière, l'architecte n'a pas besoin d'être un expert de l'IDM. En effet, il peut concevoir son système en utilisant une représentation qu'il maîtrise.

5.2.1 Métamodèle de structure du système

Afin d'élaborer notre métamodèle, nous avons effectué une analyse de domaine sur seize modèles à composants et modèles orientés services. De plus, nous avons exploité l'expérience que nous avions acquise lorsque nous avons établi le métamodèle générique de FIESTA [WLD07b, WLD07a] et les métamodèles du projet FAROS [PCW08, LWC+07]. Le métamodèle de FIESTA et le métamodèle de FAROS ont pour objectif de permettre la représentation d'un assemblage de composants indépendamment du modèle à composants utilisé (cf. section 3.1.1 et section 3.3). Nous avons donc enrichi ce métamodèle avec les concepts portant sur les interactions entre composants. Nous avons aussi étendu l'analyse de domaine sur d'autres modèles à composants et orientés services, comme SCA et OpenCOM, afin que le métamodèle de CALICO puisse modéliser un plus grand nombre de modèles. Dans cette section, nous allons détailler notre métamodèle. Les choix que nous avons faits durant la sélection des concepts sont justifiés dans la section 5.2.4.

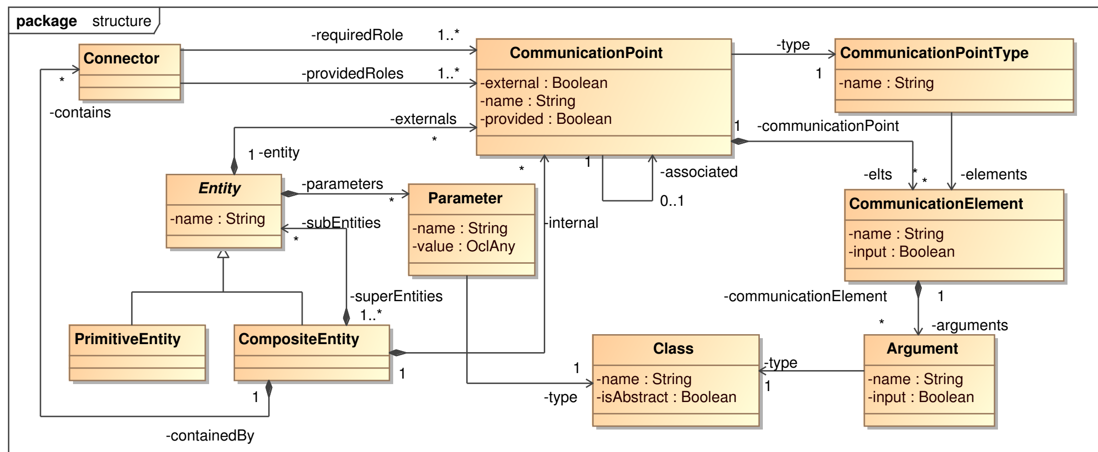

FIG. 5.1: Métamodèle de structure du système

A) Métamodèle

Le métamodèle de structure du système résultant de notre analyse de domaine est présenté dans la figure 5.1. Il contient six grands concepts architecturaux communs aux modèles à composants et aux modèles orientés services et qui sont bien connus d'un architecte logiciel. Une *entité* (`Entity`) est un élément de calcul ou de stockage de données. Elle correspond à un composant dans une architecture à composants ou à un service dans une architecture orientée services. Un *paramètre* (`Parameter`) d'une entité permet de définir une propriété de configuration de l'entité. Chaque paramètre possède un nom, un type et une valeur. Cette notion est appelée propriété ou attribut selon le modèle utilisé. Un *point de communication* (`CommunicationPoint`) est un élément d'une entité qui permet à celle-ci de communiquer avec son environnement, c'est-à-dire, les autres entités. Chaque point de communication possède un type (`CommunicationPointType`). Ce concept est appelé port ou interface dans les architectures à composants et portType dans les architectures orientées services. Un *élément de communication* (`CommunicationElement`) correspond à l'action réalisée pour communiquer entre les entités. Cette action peut être une invocation d'opération ou un envoi de messages. L'*argument* (`Argument`) est l'élément échangé entre les entités. Il peut correspondre à un objet typé ou à un message. Un *connecteur* (`Connector`) décrit les interactions entre les entités. Ce concept est équivalent aux liaisons dans les architectures à composants ou au partnerLink dans les architectures orientées services.

En outre, nous offrons la possibilité de composer hiérarchiquement les entités en fournissant le concept d'entité *composite* (`CompositeEntity`) et *primitive* (`PrimitiveEntity`). Une entité primitive est une boîte noire indivisible, tandis qu'une entité composite est constituée d'un assemblage d'entités et de connecteurs, ce concept est équivalent aux composants composites dans les architectures à composants. De plus, dans notre approche un point de communication est soit *fourni* si l'entité exporte une fonctionnalité, soit *requis* si l'entité a besoin d'importer une fonctionnalité. De même, les arguments peuvent être soit des arguments entrant dans l'entité soit sortant de l'entité.

B) Contraintes

Pour rendre explicite la sémantique du métamodèle, CALICO offre un ensemble d'attributs OCLs [Obj06b] de haut niveau qui ont un sens propre aux modèles à composants et orientés services (cf. listing 5.1). Ces attributs permettent de spécialiser le langage OCL avec des concepts dédiés aux modèles à composants et orientés services. Par exemple, les attributs

```
1 context CommunicationPoint
2 def : connectorSet : OrderedSet(Connector) =
3   if provided
4   then connectorsFromClient
5   else connectorsFromServer
6   endif

8 def : oppositePortSet : Sequence(CommunicationPoint) =
9   if provided
10  then connectorsFromClient.requiredRoles
11  else connectorsFromServer.providedRoles
12  endif

14 def : other : CommunicationPoint = oppositePortSet->first()

16 def : container : Entity = oclAsType(ecore::EObject).eContainer().oclAsType(Entity)
```

LST. 5.1: Attributs OCL portant sur le métamodèle de la structure du système

```
1 context Entity
2   inv : self.external->forAll(external)
3   inv : self.external->isUnique(name)

5 context PrimitiveEntity
6   inv : self.external.associated->excluding(null)->isEmpty()

8 context CompositeEntity
9   inv : self.internal->forAll(!external)
10  inv : self.internal->forAll(cp | cp.associated!=null and
11        self.external->includes(cp.associated) and
12        cp.associated.type=cp.type )
13  inv : self.subEntities->isUnique(name)

15 context CommunicationPoint
16  inv : self.elts->isUnique(name)

18 context CommunicationElement
19  inv : self.arguments->isUnique(name)

21 context Connector
22  inv : self.providedRoles->forAll(provided)
23  inv : self.requiredRoles->forAll(provided=false)
24  inv : self.providedRoles.container.superEntities->forAll(c|c=self.containedBy) or
25        self.providedRoles.container->forAll(c|c=self.containedBy)
26  inv : self.requiredRoles.container.superEntities->forAll(c|c=self.containedBy) or
27        self.requiredRoles.container->forAll(c|c=self.containedBy)
```

LST. 5.2: Contraintes OCL portant sur le métamodèle de la structure du système

`oppositePortSet` (lignes 8 à 12) et `other` (ligne 14) sont des attributs du point de communication et font référence aux autres points de communication qui sont connectés avec lui. Ces attributs sont rendus explicites dans tout le canevas CALICO. Ils sont manipulables par l'architecte logiciel et par le développeur de CALICO, afin de faciliter l'écriture des contraintes et de les rendre plus lisibles.

Nous avons aussi défini un ensemble d'invariants OCL, à l'aide de ces attributs, pour définir les contraintes communes à tous les modèles à composants et orientés services. Ces contraintes, offertes par CALICO, sont représentées dans le listing 5.2 Ainsi, ces invariants garantissent que les modèles décrits avec notre métamodèle respectent les propriétés communes des modèles à composants et orientés services. Par exemple, les contraintes vérifient que toutes les entités, sauf l'entité racine, sont contenues dans un seul composite et que les connecteurs connectent bien des entités contenues dans le même composite (lignes 24 à 27). Elles vérifient aussi qu'un connecteur connecte bien un point de communication fourni avec un point de communication requis (lignes 22 et 23). Nous vérifions aussi l'unicité des noms, c'est-à-dire qu'au sein d'un même composite toutes les sous-entités doivent avoir un nom unique (ligne 13) et que les noms des points de communication d'une entité doivent aussi être uniques (ligne 3).

Toutes les autres contraintes sont soit des contraintes spécifiques à une plate-forme à composants ou orientée services, soit des contraintes de niveau applicatif. Les contraintes spécifiques à une plate-forme sont écrites par un expert de la plate-forme qui ajoute le support d'une nouvelle plate-forme dans CALICO. Elles sont expliquées dans la section 5.2.2. Les contraintes applicatives sont définies par l'architecte logiciel et sont présentées dans la section 5.3.

5.2.2 Gestion des spécificités des plates-formes

Notre métamodèle de structure du système a été conçu à partir des concepts communs à différents ADLs. Cependant, chaque modèle à composants ou orienté services possède ses propres contraintes liées à la plate-forme d'exécution. Par exemple, les règles de composition entre les composants ou les services sont spécifiques à chaque modèle, comme cela a été expliqué dans la section 2.1.2. CALICO prend en compte ces contraintes spécifiques à une plate-forme grâce à son mécanisme de profil de plate-forme. Un profil encapsule les contraintes propres à une plate-forme.

A) Profil de plate-forme

Notre métamodèle a été conçu de manière à pouvoir spécifier les contraintes liées à une plate-forme. Concrètement, chacun des éléments de notre métamodèle possède une liste d'attributs capturant les propriétés spécifiques à une plate-forme. Un attribut est un couple $(clé, valeur)$. La clé et la valeur sont deux chaînes de caractères. Toutes les contraintes spécifiques à une plate-forme sont exprimées à l'aide de contraintes OCL portant sur ces attributs. De plus pour soulager l'architecte d'avoir à décrire ces contraintes spécifiques à une plate-forme, CALICO possède un système de profil de plate-forme. Un profil est un fichier créé par un expert de la plate-forme et contient l'ensemble des attributs et des contraintes spécifiques à cette plate-forme. Un profil de plate-forme contient deux catégories de contraintes. D'une part, les contraintes de composition qui correspondent aux règles de composition spécifiques à une plate-forme (cf. section 2.1.2). Ces contraintes permettent de garantir que l'architecture décrite par l'architecte est conforme aux règles de composition de la plate-forme. D'autre part, les contraintes d'implémentation qui permettent de faire le lien entre la conception et l'implémentation. Elles peuvent par exemple servir à indiquer pour chaque entité primitive la classe d'implémentation.

Globalement ce mécanisme de profil autorise un expert de la plate-forme à ajouter le support d'une nouvelle plate-forme dans CALICO. Afin d'aider l'architecte à spécifier les règles de composition du modèle, CALICO offre un langage OCL que nous avons spécialisé pour la définition des contraintes dans les modèles à composants et orientés services, grâce à des attributs OCL. Ainsi, ce mécanisme permet de garantir statiquement que l'architecture du système définie par l'architecte respecte les contraintes liées à la plate-forme cible et pourra donc être déployée sur cette plate-forme.

B) Sélection du profil de plate-forme

Grâce au mécanisme de profil de CALICO, en fonction de la plate-forme d'exécution cible choisie par l'architecte, le profil de plate-forme adéquate est automatiquement chargé. Ce mécanisme repose sur le métamodèle d'intégration de CALICO dont un sous ensemble est représenté sur la figure 5.2. Dans ce métamodèle, la méta-classe `CALICO` représente le canevas CALICO. L'association `platforms` entre la méta-classe `CALICO` et la méta-classe `Platform` symbolise l'ensemble des plates-formes d'exécution gérées par CALICO. Une plate-forme possède un nom, comme par exemple Fractal ou OpenCCM, et un profil qui correspond au nom du fichier OCL, écrit par l'expert de plate-forme, qui contient les contraintes.

De plus, le métamodèle d'intégration sert aussi à décrire le système (`System`) conçu par l'architecte. Un système contient l'entité racine (association `root`), ainsi qu'une plate-forme d'exécution cible (association `platform`).

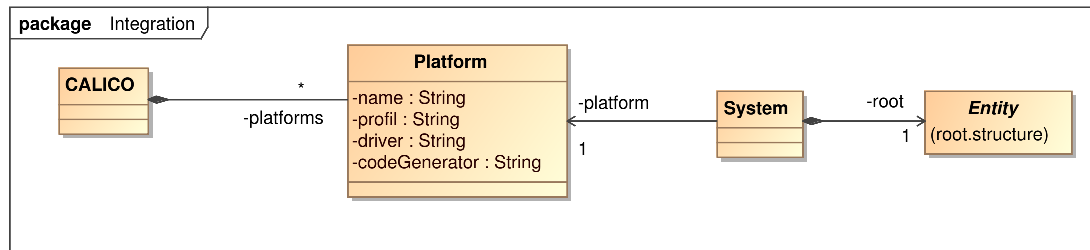

FIG. 5.2: Métamodèle d'intégration

C) Exemples

Dans la suite de cette section, nous présentons cinq exemples de profil de plate-forme correspondant aux cinq plates-formes décrites dans la section 2.1.2.

Profil OpenCOM. Le listing 5.3 est un extrait du profil de plate-forme OpenCOM. L'invariant de la ligne 2 est une contrainte d'implémentation. Il impose à l'architecte de définir pour chaque composant OpenCOM un attribut dont la clé est `impl-class` et dont la valeur correspond au nom de la classe d'implémentation du composant.

Les lignes 4 à 9 correspondent aux règles de composition du modèle OpenCOM (cf. section 2.1.2). Par exemple, l'invariant de la ligne 6 contraint l'architecte à connecter uniquement un point de communication requis avec un point de communication fourni de même type. Celui de la ligne 9 impose que chaque point de communication requis doit être connecté.

```
1 context PrimitiveEntity
2   inv: self.properties->exists(p:Property | p.name='impl-class')

4 context Connector
5   inv: self.requiredRoles->size()=1 and self.providedRoles->size()=1
6   inv: self.requiredRoles->first().type=self.providedRoles->first().type

8 context CommunicationPoint
9   inv: not self.provided implies self.connectorSet->size()=1
```

LST. 5.3: Profil OpenCOM

Profil Fractal. De la même manière, le profil de la plate-forme Fractal est représenté dans le listing 5.4. L'invariant de la ligne 2 est une contrainte d'implémentation qui permet d'associer pour chaque composant Fractal primitif le nom de la classe d'implémentation.

Les invariants des lignes 4 à 16 correspondent aux règles de composition du modèle Fractal (cf. section 2.1.2). Le mécanisme d'attribut fourni par le métamodèle de CALICO permet à un expert de la plate-forme Fractal de décrire les spécificités de la plate-forme. Par exemple, la contrainte des lignes 5 à 8 autorise l'architecte à spécifier la contingence d'un point de communication en déclarant un attribut qui a pour clé la chaîne `contingency`. La contrainte impose aussi que la valeur de l'attribut soit `optional` ou `mandatory`, ce qui correspond respectivement à déclarer un port Fractal optionnel ou obligatoire. Cet attribut est ensuite utilisé dans la contrainte des lignes 9 à 12 pour imposer que si un point de communication est déclaré `mandatory`, alors il doit obligatoirement être connecté avec un autre point de communication. De plus, la puissance d'expression d'OCL permet aussi à l'expert de la plate-forme de spécifier le comportement par défaut. Dans notre exemple, s'il n'y a aucun attribut `contingency` alors, par défaut, le point de communication est `mandatory`.

```
1 context PrimitiveEntity
2   inv: self.properties->exists(p:Property | p.name='impl-class')

4 context CommunicationPoint
5   inv: self.properties->exists(p | p.name='contingency') implies
6        self.properties->exists(p | p.name='contingency' and
7        (p.oclAsType(SimpleProperty).value='optional' or
8        p.oclAsType(SimpleProperty).value='mandatory'))
9   inv: self.properties->exists(p | p.name='contingency' and
10       p.oclAsType(SimpleProperty).value='mandatory') or
11       not self.properties->exists(p | p.name='contingency') implies
12       self.connectorSet->size()=1

14 context Connector
15  inv: self.requiredRoles->size()>=1 and self.providedRoles->size()=1
16  inv: self.requiredRoles->forAll(cp | cp.type=self.providedRoles->first().type)
```

LST. 5.4: Profil Fractal

Profil OpenCCM. Le listing 5.5 correspond au profil de la plate-forme OpenCCM. Les lignes 15 et 19 sont des contraintes d'implémentation. Elles imposent à l'architecte de spécifier un nom de paquetage et une `home` pour chaque composant OpenCCM.

Les autres contraintes correspondent aux règles de composition (cf. section 2.1.2). Par exemple, les deux contraintes des lignes 2 à 7 imposent à l'architecte de spécifier si un point de communication fourni est une facette ou une source d'événements et si un point de communication requis est un réceptacle ou un puits d'événements. La contrainte de la ligne 16 interdit l'utilisation des composants composites, excepté pour l'entité racine, étant donné que le modèle à composants OpenCCM est un modèle plat.

```
1 context CommunicationPoint
2   inv: self.provided implies self.properties->exists(p:Property | p.name='type' and
3        (p.oclAsType(SimpleProperty).value='facette' or
4        p.oclAsType(SimpleProperty).value='evenSource'))
5   inv: not self.provided implies self.properties->exists(p:Property | p.name='type' and
6        (p.oclAsType(SimpleProperty).value='receptacle' or
7        p.oclAsType(SimpleProperty).value='enventSink'))
8   inv: not self.provided implies self.connectorSet->size()=1

10 context Connector
11  inv: self.requiredRoles->size()=1 and self.providedRoles->size()=1
12  inv: self.requiredRoles->first().type=self.providedRoles->first().type

14 context CompositeEntity
15  inv: self.properties->exists(p:Property | p.name='package')
16  inv : self.superEntities->excluding(null)->isEmpty()

18 context PrimitiveEntity
19  inv : self.properties->exists(p:Property | p.name='home')
```

LST. 5.5: Profil OpenCCM

Profil FraSCAti. Le mécanisme de profil de plate-forme est suffisamment souple pour permettre aussi la définition de différents profils pour les différentes versions d'une même plate-forme. Il est ainsi possible de capturer les différentes fonctionnalités offertes par les différentes versions de la plate-forme. Par exemple, le listing 5.6 correspond au profil de la version 0.4 de la plate-forme FraSCAti. Les contraintes des lignes 2 à 10 et de la ligne 16 sont des contraintes d'implémentation. La contrainte de la ligne 2 permet à un architecte de spécifier, pour chaque composant SCA primitif, le nom de la classe d'implémentation. Celles des lignes 3 à 5 imposent à l'architecte de définir le type d'implémentation du composant primitif. Dans cet exemple, la version de la plate-forme de FraSCAti supporte seulement les composants écrits en Java ou en Fractal. L'ajout du code `p.oclAsType(SimpleProperty).value='spring'` suffit pour spécifier

```
1 context PrimitiveEntity
2   inv: self.properties->exists(p:Property | p.name='impl-class')
3   inv: self.properties->exists(p:Property | p.name='implementationType' and
4        (p.oclAsType(SimpleProperty).value='java' or
5        p.oclAsType(SimpleProperty).value='fractal'))

7 context Connector
8   inv: self.properties->exists(p | p.name='bindingType') implies
9        self.properties->exists(p:Property | p.name='bindingType' and
10       (p.oclAsType(SimpleProperty).value='rmi' or
11       p.oclAsType(SimpleProperty).value='ws'))
12  inv: self.providedRoles->size()=1
13  inv: self.requiredRoles->size()>=1
14  inv: self.requiredRoles->forAll(cp | cp.type=self.providedRoles->first().type)

16 context CommunicationPoint
17  inv: self.external implies self.properties->exists(p:Property | p.name='implementationType')
18  inv: not self.provided implies self.connectorSet->size()=1
```

LST. 5.6: Profil FraSCAti

que la version de la plate-forme supporte aussi les implémentations Spring.

Profil des services WEB. Le profil de plate-forme peut aussi servir pour spécifier des informations qui seront utilisées pour l'exécution de l'application. Par exemple, dans le cas des architectures orientées services, chaque service est accessible via une adresse internet et un numéro de port. Un expert de la plate-forme de services WEB peut indiquer que ces informations sont indispensables en créant un attribut `hostname` et un attribut `port` (cf. lignes 6 à 10 du listing 5.7). De cette manière, l'administrateur du système pourra ultérieurement spécifier ces informations en mettant directement à jour le modèle de structure du système.

```
1 context Entity
2   inv: self.external->select(p:CommunicationPoint | p.provided=true)->size()<=1

4 context PrimitiveEntity
5   inv: self.properties->exists(p:Property | p.name='impl-class')
6   inv: (self.external->select(p:CommunicationPoint |
7        p.provided=true)->size()=1) implies (self.properties->exists(p:Property | p.name='hostname'))
8   inv: (self.external->select(p:CommunicationPoint |
9        p.provided=true)->size()=1) implies (self.properties->exists(p:Property | p.name='port'))

11 context Connector
12   inv: self.requiredRoles->size()=1 and self.providedRoles->size()=1
13   inv: self.requiredRoles->first().type=self.providedRoles->first().type
```

LST. 5.7: Profil des services Web

5.2.3 Exemple

Cette section illustre l'utilisation du métamodèle de structure du système pour représenter l'architecture du système du Dossier Médical Personnel (DMP) présenté dans la section 4.2. La figure 5.3 est un extrait du modèle de structure du système. Elle modélise les éléments architecturaux `GlobalSearch` et `MedicalServer`. Chaque élément architectural correspond à une entité. Les fonctionnalités requises et fournies sont respectivement modélisées par des points de communication requis et fournis. Les liens entre les fonctionnalités correspondent à des connecteurs. Dans cet exemple, l'entité `MedicalServer` possède un point de communication fourni `getData`. Ce point de communication contient un élément de communication `getPicture` qui prend, en entrée, un argument `URL` et, en sortie, un argument `data` qui est de type `MedicalInformation`.

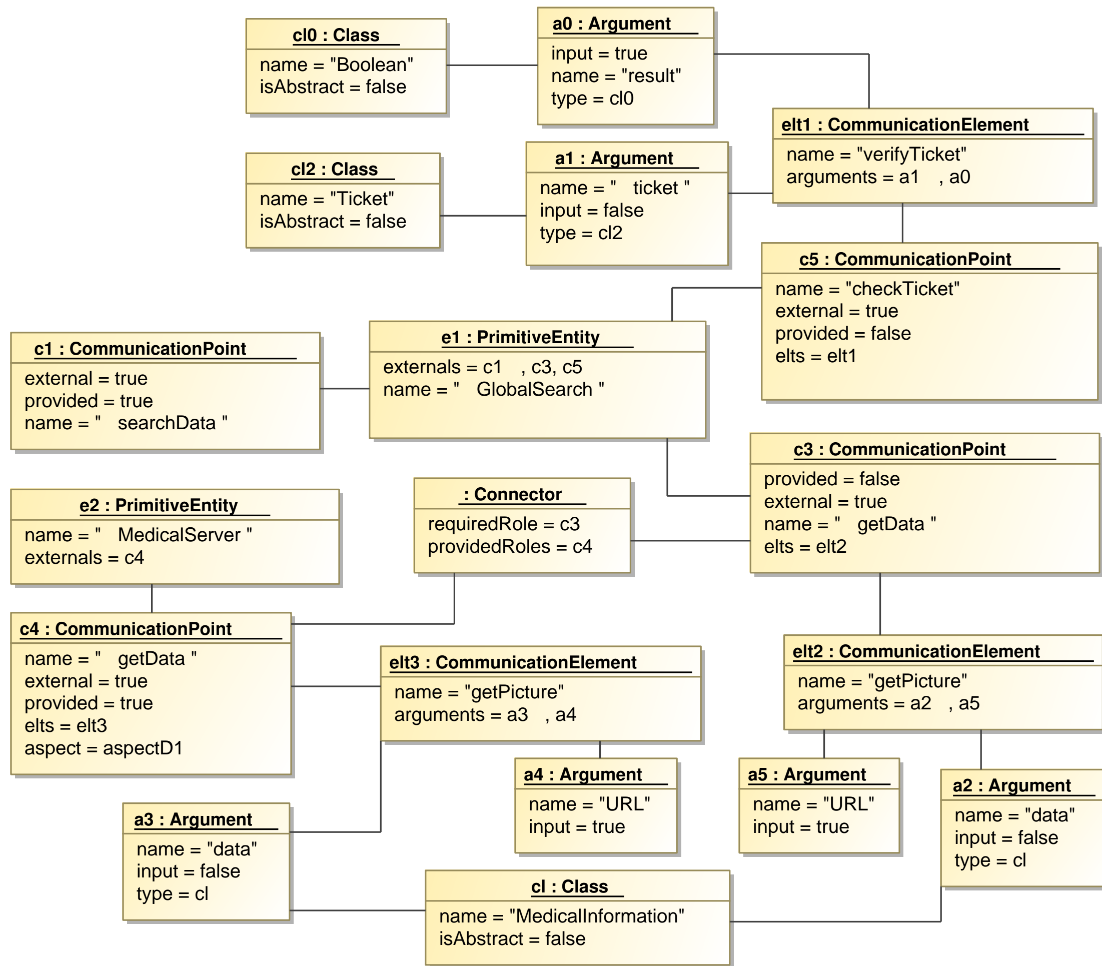

FIG. 5.3: Extrait du modèle de structure du système DMP

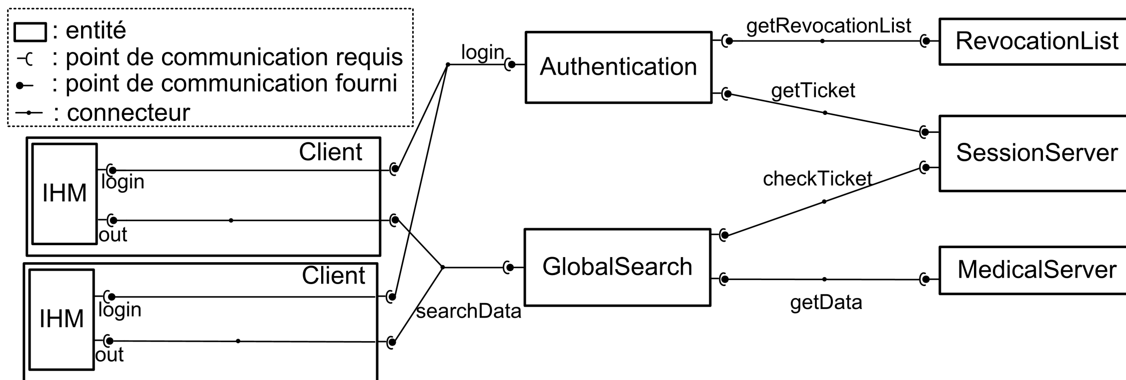

FIG. 5.4: Représentation graphique du modèle de la structure du système DMP

Conscient qu'un modèle de structure du système peut rapidement devenir volumineux et difficilement lisible, CALICO propose une surcouche graphique permettant à un architecte logiciel d'utiliser une représentation graphique qu'il connaît bien (cf. section 2.1.2 de l'état de l'art). La figure 5.4 est une représentation graphique du système DMP utilisant le formalisme graphique of-

fert par CALICO. Dans cet exemple, un formalisme graphique ressemblant à celui de OpenCCM est utilisé.

En outre, à n'importe quel moment, l'architecte peut raffiner son modèle en y ajoutant les informations spécifiques à la plate-forme d'exécution choisie afin de rendre possible l'exécution du système. Par exemple, il peut raffiner l'entité `Authentication` avec les informations spécifiques au modèle à composants SCA. Pour spécifier que l'implémentation du composant `Authentication` sera faite en Java et que les points de communication seront écrits en Java, l'architecte a besoin d'ajouter une propriété de clé `implementationType` et de valeur `java` dans l'entité `Authentication` et dans chaque point de communication de cette entité.

5.2.4 Discussion

Notre métamodèle est similaire aux ADLs génériques, comme Acme [GMW00] et xADL [DdHT01], cependant ces modèles n'ont pas imposé de sémantique forte aux points de communication qui restent donc flous. Certes, cette solution permet à ces ADLs d'être très génériques, mais empêche malheureusement l'application non ambiguë d'analyse des interactions. C'est donc pour cette raison que nous avons fait des choix plus stricts lors de la définition des points de communication. Nous avons besoin que notre métamodèle explicite clairement les interactions. Nous détaillons dans la suite de cette section chacun de nos choix.

Dans notre métamodèle, un point de communication est unidirectionnel, c'est-à-dire qu'il est soit requis, soit fourni. Nous avons fait ce choix car il couvre la plus grande majorité des modèles à composants ou orientés services. Ce concept de producteur/consommateur est un principe de base des modèles à composants. Par exemple, les modèles à composants comme Fractal [BCS04], SCA [BII+07], SOFA [PV02], OpenCOM [CBG+04] ou OpenCCM [Obj02b] rendent explicite cette notion de requis/fourni, même si elle est nommée différemment. Unicon [Zel96] est un modèle à composant qui supporte 14 types de points de communication. Ce modèle n'explicite pas directement cette notion, cependant fondamentalement il existe des types de points de communication qui sont requis et d'autres qui sont fournis. Par exemple, le point de communication `RPCCall` correspond à une invocation de procédure. Il a donc une sémantique similaire à un point de communication requis. Le point de communication `RPCDef` correspond à une définition d'une procédure. Il a donc une sémantique similaire à un point de communication fourni.

D'autre part, une minorité d'ADLs ont des points de communication bidirectionnels. Cette notion permet des relations entre deux composants qui sont plus complexes qu'une simple relation client/server. En effet, elle permet d'exprimer que le serveur ne peut fournir une fonctionnalité au client que si, en retour, le client lui offre une fonctionnalité. Dans notre approche nous ne fournissons pas directement ce concept, mais il est possible d'avoir le même pouvoir d'expression en créant deux points de communication unidirectionnels (un requis et un fourni), et en ajoutant des contraintes sur ces points de communication dans un profil (cf. section 5.2.2). Les mêmes choix ont été faits par le dernier ADL générique que nous connaissons [FDH08].

Nous avons ensuite décidé qu'un point de communication contient des éléments de communication avec des arguments en entrée ou en sortie. Cette modélisation permet de décrire, à la fois, les communications par appel de fonction, et les communications par envoi de messages, ce qui correspond à la plus grande majorité des communications existantes dans les ADLs.

En outre, nous n'avons pas repris les concepts spécifiques à un modèle, comme par exemple le concept de composant partagé qui est unique au modèle Fractal. En effet, ces spécificités uniques sont incompatibles avec la majorité des algorithmes d'analyses existants. Par exemple, puisque la très grande majorité des travaux d'analyses existants ne prend en charge que des architectures arborescentes, c'est-à-dire que l'algorithme fonctionne sur des arbres et non des graphes, alors ces travaux ne sont pas applicables dans le cas des composants partagés. Nous avons donc choisi de ne pas inclure les concepts uniques à un modèle dans CALICO afin de permettre une intégration directe d'un maximum d'outils existants.

Globalement, nous avons créé notre métamodèle générique pour décrire la structure d'un système en nous reposant sur différents travaux proposant un ADL générique, comme Acme [GMW00], xADL [DdHT01] et FIESTA [WLD07b], ainsi que sur des travaux plus récents

qui ont unifié les modèles à composants et les modèles orientés services [BII+07, ABH+07]. Notre objectif n'est pas de définir un autre modèle à composants ou orienté services, mais d'isoler un sous ensemble représentatif de concepts architecturaux permettant la conception d'un système et la modélisation des interactions qui est requise pour la vérification de la cohérence du système.

5.3 Spécification des propriétés applicatives

Une fois que l'architecte a défini la structure de son architecture, il a besoin de spécifier les diverses propriétés applicatives de son système. Nous avons mis en évidence les propriétés applicatives que l'architecte tient à saisir et à vérifier au niveau architectural dans l'état de l'art.

Dans cette section, nous présentons nos différents métamodèles qui permettent la spécification des différentes propriétés applicatives de manière indépendante de toute plateforme. La section 5.3.1 explique le paradigme que nous avons choisi pour permettre l'expression des différentes propriétés applicatives. Ensuite, les sections 5.3.2 à 5.3.5 décrivent les métamodèles de CALICO qu'un architecte peut utiliser pour spécifier respectivement les propriétés structurelles, comportementales, de flot de données et de QdS. Au cours de cette section, chaque métamodèle est illustré avec un extrait de l'exemple du système DMP, présenté dans la section 4.2.

5.3.1 Paradigme choisi pour la spécification

Dans notre approche, nous désirons utiliser un formalisme uniforme et générique qui permet la composition des spécifications. Après investigation des paradigmes existants, nous avons opté pour le paradigme hypothèse/garantie [AL93].

Dans ce paradigme, une spécification est toujours associée à un *participant* P telle que :

- $hypthèse(P)$ rend explicite les propriétés applicatives que P exige de la part de son environnement, c'est-à-dire tout ce qui interagit avec P.
- $garantie(P)$ explicite les propriétés applicatives que P offre à son environnement.

Sur la base des spécifications d'$hypothèse$ et de $garantie$, il sera possible de déterminer si l'ensemble de l'architecture répond à toutes ces propriétés. Ainsi, les spécifications sont composées telles que :

$$hypothèse(P) \rightarrow garantie(P),$$

où l'opérateur $\rightarrow$ est une forme logique, qui exprime le fait que si l'hypothèse P est vraie alors la garantie P est vraie [AL93].

Ce paradigme d'hypothèse/garantie a souvent été utilisé dans le cadre de la QdS [IBM03, FK98]. Nous généralisons l'utilisation de cette approche à la spécification des propriétés structurelles, comportementales et du flot de données. De cette manière, un architecte logiciel peut manipuler les quatre catégories de propriétés applicatives au sein de son architecture : structurelle pour limiter les spécifications de la structure de son architecture ; comportementale pour décrire la synchronisation des entités, c'est-à-dire leur flot de contrôle ; flot de données pour restreindre les valeurs des données échangées ; et de QdS pour capturer les propriétés extra-fonctionnelles, telles que le temps de réponse. Toutes ces caractéristiques sont spécifiques à chaque application et sont composées et analysées afin de vérifier que l'application répond bien à toutes ces propriétés souhaitées.

Dans la suite de cette section, nous illustrons l'utilisation de ce paradigme à l'aide de quatre exemples de métamodèles correspondant respectivement aux quatre catégories de propriétés applicatives. Lors de la définition de ces métamodèles, nous avons toujours choisi de proposer des métamodèles les plus génériques possibles en nous reposant sur l'existant. Cependant, CALICO n'est pas limité à ces quatre métamodèles. En effet, CALICO est extensible et autorise la définition de nouveaux métamodèles par un expert du domaine. Les seules contraintes sont que d'une part,

le nouveau métamodèle doit être basé sur le paradigme hypothèse/garantie et que d'autre part il doit être couplé avec le métamodèle de structure du système.

5.3.2 Spécification structurelle

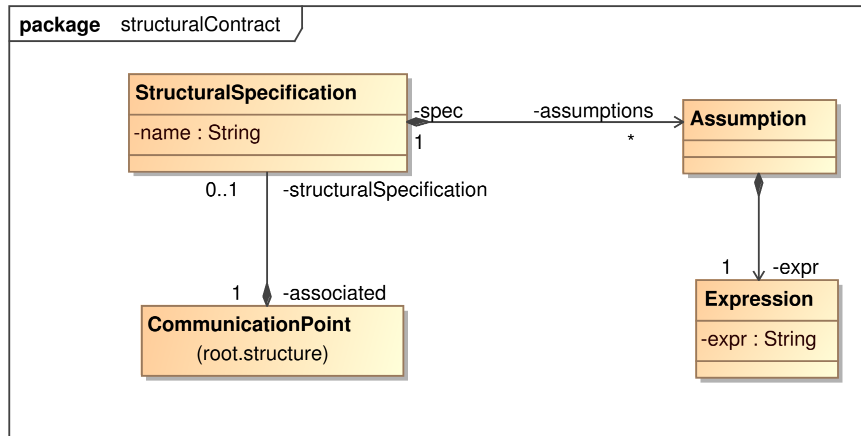

FIG. 5.5: Métamodèle des spécifications structurelles

Les propriétés structurelles permettent à l'architecte d'exprimer les exigences structurelles de son application. Pour spécifier ces propriétés, l'architecte définit des spécifications structurelles et il les associe aux points de communication des entités, comme le montre la figure 5.5. Les spécifications sont exprimées à l'aide du langage OCL [Obj06b], qui est un langage de contrainte bien connu des architectes logiciels. Nous limitons l'utilisation de l'OCL à ses invariants. Le contexte de l'invariant OCL correspond implicitement au point de communication qui porte la spécification. Afin de décrire plus facilement ses contraintes, l'architecte peut aussi réutiliser les attributs OCL, présentés dans la section 5.2.1, comme par exemple, l'attribut `other` qui fait référence aux autres points de communication qui sont connectés au point de communication qui porte la contrainte. L'architecte déclare uniquement les exigences structurelles, ce qui correspond à ajouter des expressions d'hypothèse en vue de contraindre les connexions possibles d'un point de communication. Toutes les garanties structurelles sont directement explicites dans le modèle de structure du système.

Exemple. Pour exprimer les propriétés structurelles présentées dans la section 4.2.2, c'est-à-dire que le point de communication `getTicket` de l'entité `SessionServer` ne doit être utilisé

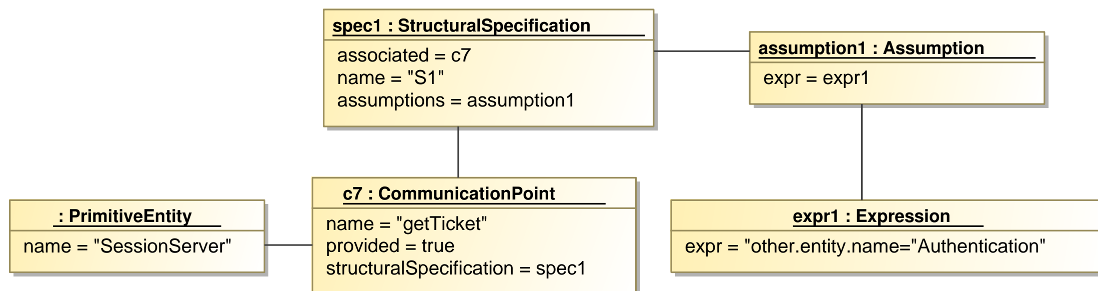

FIG. 5.6: Modèle de la spécification structurelle `S1`

que par l'entité `Authentication`, l'architecte logiciel peut ajouter la spécification structurelle suivante `S1` sur le point de communication `getTicket` :

```
S1 : other.entity.name=''Authentication''.
```

La figure 5.6 représente le modèle de la spécification structurelle `S1`. Une spécification structurelle nommée `S1` est attachée au point de communication `getTicket` de l'entité `SessionServer`. La spécification a pour expression la contrainte OCL : `other.entity.name=''Authentication''`.

5.3.3 Spécification comportementale

Afin d'être le plus générique possible, la spécification du comportement repose sur la notion de graphe de flot de contrôle [All70]. Cette notion est commune aux différents langages de spécification du comportement existant dans les modèles à composants et dans les modèles orientés services, comme CSP [Hoa85], FSP [Mag99], SFSP [Bar05], UML [Obj07b] et BPMN [Obj07a].

Ces spécifications sont définies à l'aide du métamodèle des spécifications comportementales de la figure 5.7. Une spécification est associée aux points de communication. Elle peut être primitive ou composite. Une spécification primitive est écrite par l'architecte logiciel et permet à ce dernier d'exprimer le comportement d'une entité architecturale en décrivant les flots de contrôles entrants et sortants des points de communication. Une spécification composite résulte d'une composition et décrit le comportement d'un assemblage d'entités. Une spécification peut être vue comme une expression hypothèse/garantie [MG07]. La garantie décrit le comportement que l'entité respecte et l'hypothèse exprime le comportement que l'entité exige.

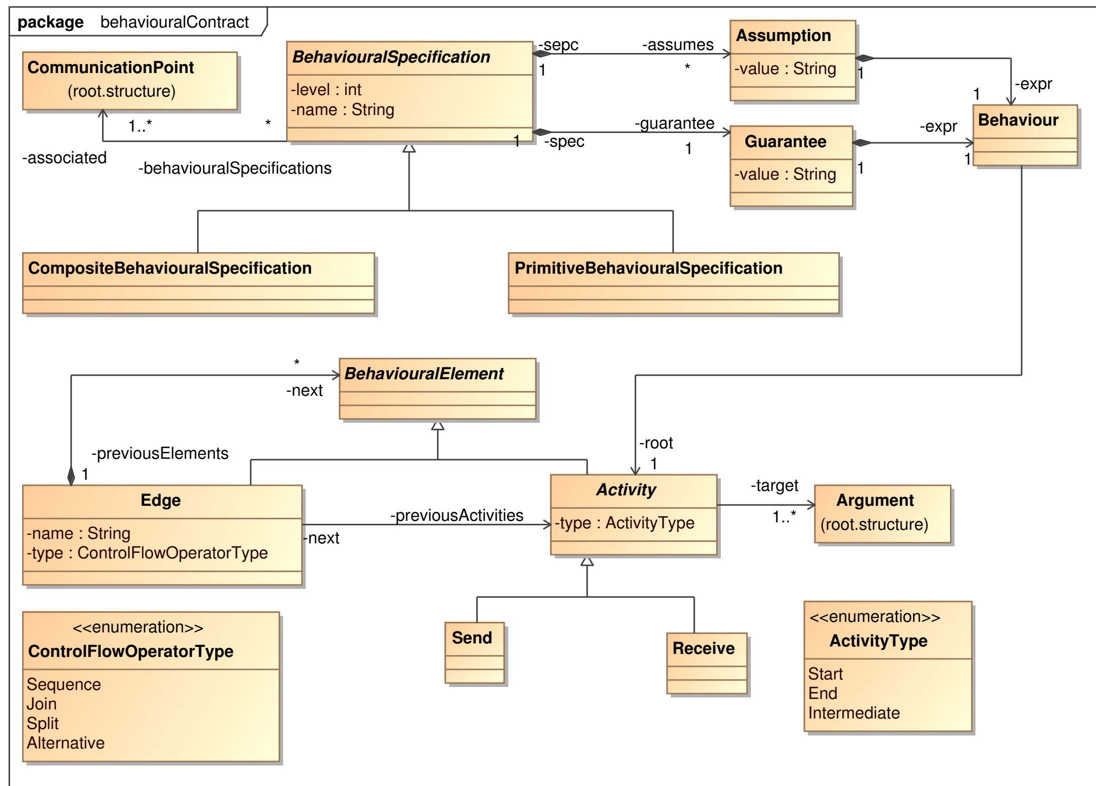

FIG. 5.7: Métamodèle des spécifications comportementales

Le comportement est décrit à l'aide d'un graphe de flot de contrôle. Un graphe de flot de contrôle est composé d'*arcs* orientés et d'*activités*. Un arc correspond à un opérateur de flot de contrôle non structuré [RtHvdAM07], qui peut être de quatre types : la séquence, l'alternative, la fusion et la séparation du flot de contrôle. Ces opérateurs sont similaires aux noeuds de contrôle définis dans la superstructure d'UML2 : `DecisionNode`, `ForkNode` et `MergeNode` [Obj07b]. Les activités sont les noeuds du graphe. Ils correspondent à l'envoi ou à la réception d'arguments, ils représentent les états dans lesquels une entité envoie ou attend des messages. Dans notre approche, les graphes de flot de contrôle sont acycliques, c'est-à-dire que pour chaque noeud, il existe exactement un chemin partant de l'origine du graphe et allant à ce noeud. Donc notre graphe est sensible au contexte car il est possible, à partir de chaque noeud du graphe, de retrouver l'historique des messages échangés, c'est-à-dire la trace qui a conduit l'entité dans cet état.

Notre métamodèle est découplé du moyen permettant sa description. Par exemple, l'architecte logiciel peut décrire ces spécifications en concevant directement un modèle de flot de contrôle conforme au métamodèle de la figure 5.7. Sinon il peut aussi utiliser un langage dédié similaire à SFSP ou alors il peut élaborer un diagramme graphique de processus similaire à BPMN.

Notre métamodèle est fortement couplé avec le métamodèle de structure du système. Ce couplage nous permet d'accroître la cohérence entre ces deux métamodèles. Il repose sur la définition d'un ensemble de contraintes OCL qui est automatiquement vérifié. Par exemple, nous avons défini des invariants OCL qui empêchent l'architecte de déclarer une activé d'envoi de message portant sur un argument entrant dans un point de communication fourni.

Exemple. Nous n'illustrons que le flot de contrôle qui implique les entités `Client`, `GlobalSearch` et `MedicalServer`. Nous exprimons les hypothèses et les garanties correspondant aux propriétés présentées dans la section 4.2.2. Par exemple, la garantie comportementale `B1` de l'entité `MedicalServer` associée au point de communication `getData` est représentée avec le modèle comportemental de la figure 5.8. La spécification comportementale primitive, nommé `B1`, est attachée au point de communication `getData` de l'entité `MedicalServer`. Elle contient une garantie exprimant que l'élément de communication `getPicture` reçoit l'URL d'une donnée médicale et retourne ensuite (`sequence`) en réponse la donnée `data`. Cette spécification peut aussi être écrite avec le langage SFSP :

```
B1 : ?getPicture.URL ; !getPicture.data$
```

Pour rappel, le point d'exclamation exprime un envoi de message et le point d'interrogation exprime la réception d'un message. CALICO propose aussi un formalisme de représentation

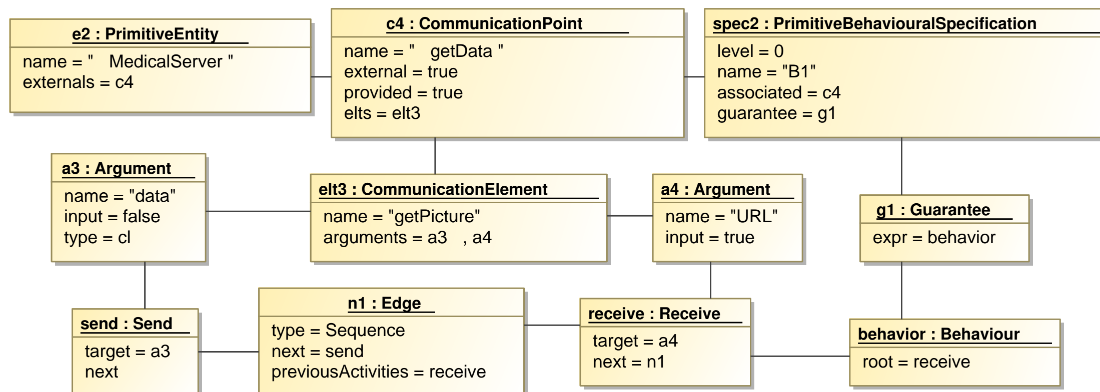

FIG. 5.8: Modèle de la spécification comportementale `B1`

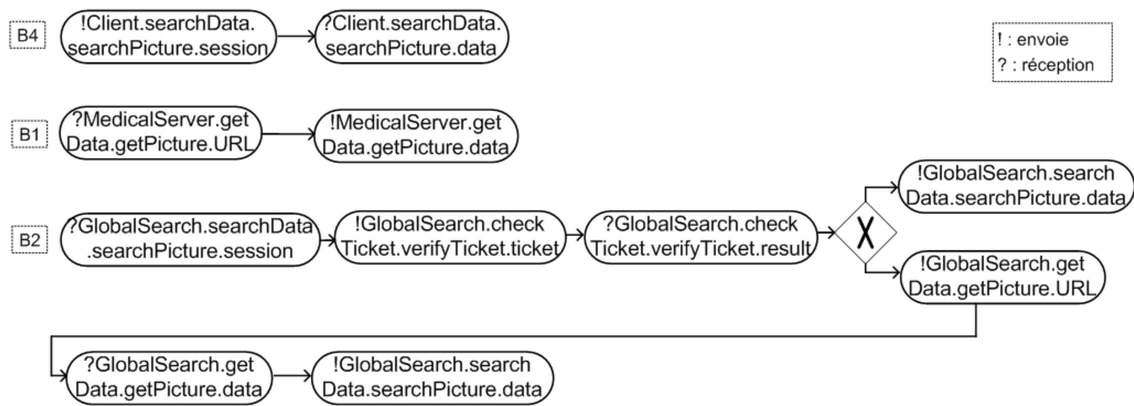

FIG. 5.9: Représentation graphique des spécifications comportementales B1, B2 et B4

graphique. La figure 5.9 est la représentation graphique de la spécification comportementale B1. Dans cet exemple, un formalisme graphique similaire à BPMN est utilisé.

De même, l'architecte exprime la garantie comportementale de l'entité GlobalSearch en attachant une spécification B2, à la fois, sur les points de communication searchData, checkTicket et getData. L'entité GlobalSearch attend de recevoir une demande de recherche de dossier médical (session). Après, en séquence, l'entité envoie le ticket de session (ticket) et attend le résultat de la validation de la session result. Ensuite l'entité retourne soit une donnée data nulle si le ticket n'a pas été validé, soit envoie la requête de recherche de la donnée médicale (URL), attend la réponse (data) et renvoie cette réponse data au client. Ce comportement est représenté graphiquement en bas de la figure 5.9 en utilisant une notation similaire à BPMN. La description SFSP de la spécification est la suivante :

```
B2 : ?searchData.searchPicture.session ; !checkTicket.verifyTicket.ticket ;
     ?checkTicket.verifyTicket.result$ ; (!searchData.searchPicture.data$ |
     !getData.getPicture.URL ; ?getData.getPicture.data$ ;
     !searchData.searchPicture.data$)
```

De la même façon, l'architecte peut spécifier le comportement de l'entité Client en attachant une spécification comportementale B4, comme le montre la figure 5.9. L'entité Client envoie d'abord les données de session (session) qui contiennent le ticket de sessions et l'URL de la donnée médicale recherchée (URL). En réponse, l'entité attend de recevoir la donnée (data).

L'entité MedicalServer exige que l'élément de communication verifyTicket du point de communication checkTicket de l'entité SessionServer soit appelé avant l'appel de l'élément de communication getPicture du point de communication getData. Cela peut être spécifié avec l'hypothèse B3 suivante attachée sur le point de communication getData de l'entité MedicalServer.

```
B3 : ?SessionServer.checkTicket.verifyTicket.ticket;*;
     ?MedicalServer.getData.getPicture.URL
```

5.3.4 Spécification du flot de données

Les propriétés de flot de données permettent à l'architecte de restreindre les valeurs des arguments d'entrée et de sortie dans le flot. Il existe beaucoup d'applications dans le domaine de la diffusion d'information qui sont fondées sur ce principe de flot de données, comme le DMP. Pour raisonner sur ces types d'applications, il est nécessaire de pouvoir spécifier des propriétés de flot de données. Or, très peu de travaux sur les modèles à composants et les modèles orientés services prennent en charge ces propriétés. Cependant, ces propriétés sont très répandues dans

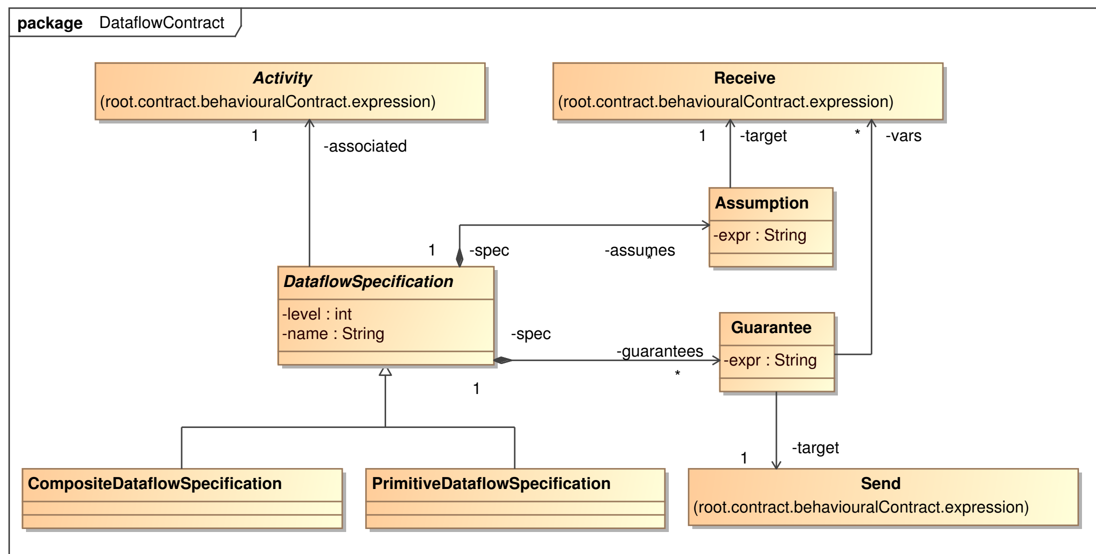

FIG. 5.10: Métamodèle des spécifications de flot de données

le domaine des analyses de programmes [Kil73]. Nous avons donc basé notre proposition sur ces travaux.

Pour spécifier ces propriétés, nous avons étendu le graphe de flot de contrôle. Comme nous pouvons le voir sur le métamodèle de la figure 5.10, une activité du graphe de flot de contrôle peut être associée à une spécification de flot de données. Les spécifications peuvent être soit primitives si elles sont directement écrites par l'architecte, soit composites si elles résultent d'une analyse de composition. Ces spécifications sont des hypothèses quand elles s'appliquent aux arguments d'entrées et sont des garanties si elles contraignent les arguments de sorties. Dans notre implémentation, nous avons choisi d'utiliser le formalisme proposé par Maxima [max09], qui est un outil de calcul symbolique, pour décrire ces spécifications.

Exemple. Cette section illustre comment un architecte exprime les propriétés de flot de données du système DMP. Pour décrire les spécifications de flot de données, l'architecte doit enrichir le modèle de flot de contrôle des entités de son système, schématisé sur la figure 5.11.

Pour décrire le profil de l'entité `Client` présentée dans la section 4.2.2, l'architecte peut ajouter la spécification de flot de données `D1`, en créant le modèle de spécification de flot de données, comme sur la figure 5.12. Pour réaliser cet objectif, il doit d'abord exprimer le flot de contrôle du point de communication `searchData` de l'entité `Client`. Dans cet exemple, le point de communication `searchData` envoie une `session` qui contient l'URL de la donnée médicale, puis récupère la donnée médicale `data`. Ensuite, l'architecte peut enrichir le flot de contrôle avec les spécifications de flot de données. Concrètement, il attache à l'activité de réception de la donnée médicale, la spécification de flot de données `D1`. Cette spécification possède une hypothèse portant sur l'activité de réception dont l'expression est la suivante :

```
D1 : searchPicture.data.size<10Mb;
```

Cette hypothèse exprime que les données médicales doivent avoir une taille inférieure à 10 mégaoctets.

L'architecte logiciel doit aussi ajouter une garantie `D2` sur le point de communication `getData` de l'entité `MedicalServer` exprimant le fait que l'entité `MedicalServer` renvoie toujours des images inférieures à 20 gigaoctets, qui est la limite maximale de la base de données (cf. figure 5.11).

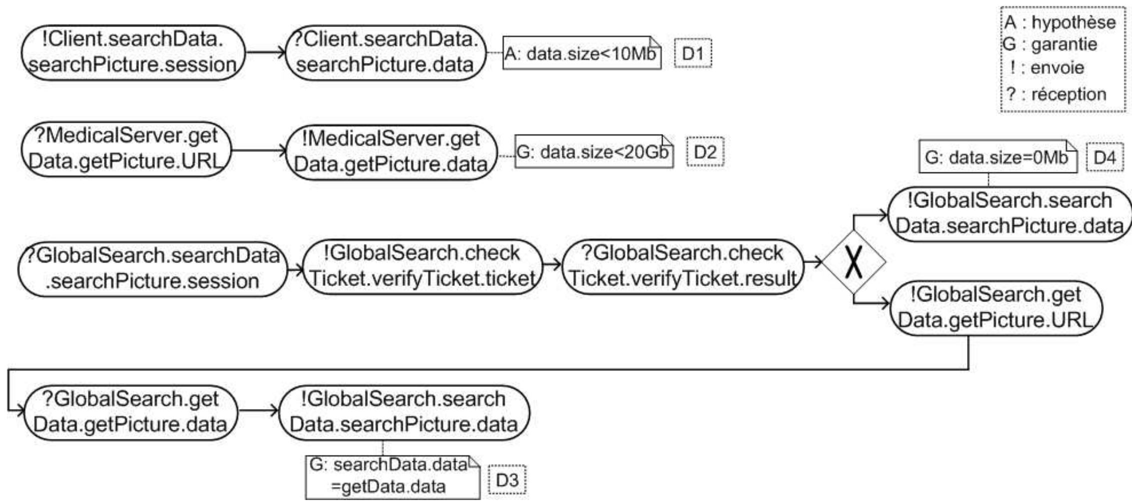

FIG. 5.11: Réprésentation graphique des spécifications de flot de données D1 à D4

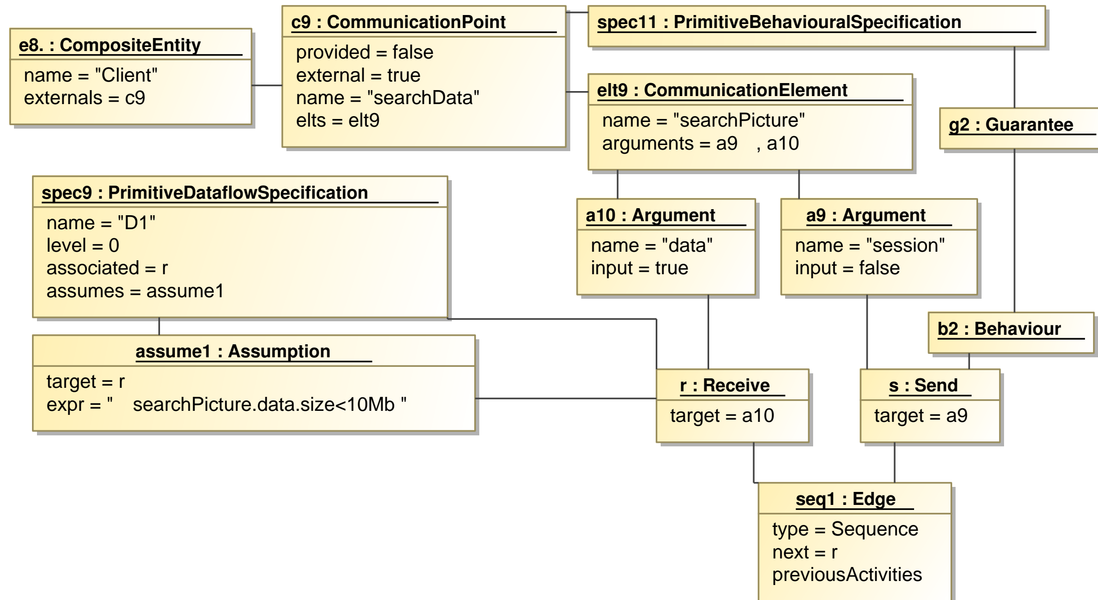

FIG. 5.12: Modèle de la spécification de flot de données D1

```
D2 : getPicture.data.size<20Gb;
```

Finalement, l'architecte spécifie que l'entité `GlobalSearch` ne modifie pas les données qu'elle reçoit grâce à la garantie `D3` attachée au point de communication `searchData` de l'entité `GlobalSearch` (cf. figure 5.11) :

```
D3 : searchData.searchPicture.data=getData.getPicture.data;
```

Il décrit aussi, à l'aide d'une garantie `D4`, que dans la branche d'exécution issue de l'échec de la validation du ticket de session, la taille de la donnée médicale renvoyée est nulle.

5.3.5 Spécification de qualité de service

Les propriétés de QdS englobent une très large variété de propriétés non fonctionnelles. Elles incluent le domaine de la sécurité, de la performance, de la maintenance et de la disponibilité du système. Chaque propriété est exprimée avec son propre langage et possède sa propre métrique. Afin d'exprimer les propriétés de QdS de manière générique, nous avons identifié les concepts communs existants dans différents langages de spécification de la QdS, comme QML [FK98], WSLA [IBM03] et WS-agreement [For07].

Le résultat de cette analyse a conduit à la définition du métamodèle de la figure 5.13. Dans ce métamodèle, une spécification de QdS est représentée par la métaclasse `QoSSpecification`. Elle est associée à une activité du flot de contrôle. Elle peut être soit primitive si elle est spécifiée par un architecte, soit composite si elle résulte d'une composition. Une spécification de QdS peut être vue comme une expression hypothèse/garantie. Elle est une hypothèse si elle correspond à une pré-condition portant sur la valeur d'une propriété non fonctionnelle et c'est une garantie si elle correspond à une post-condition. La syntaxe d'une expression de QdS est similaire à une expression QML, c'est-à-dire que c'est une expression booléenne manipulant des valeurs entières, des réelles et des chaînes de caractères. Les opérateurs de comparaison supportés sont $<, >, \leq, \geq, =, \neq$.

Afin d'être extensible et de pouvoir prendre en compte cet ensemble non borné de propriétés extra-fonctionnelles, nous avons introduit le concept de type de QdS (cf. métamodèle de la figure 5.13). Chaque type de QdS correspond à une catégorie de propriétés de QdS, comme la charge CPU en pourcentage ou la quantité de mémoire disponible en mégaoctet. Chaque spécification de QdS est associée à un type de QdS. Nous fournissons un ensemble de type de QdS prédéfini et notre métamodèle permet la définition de nouveaux types selon les besoins de l'architecte.

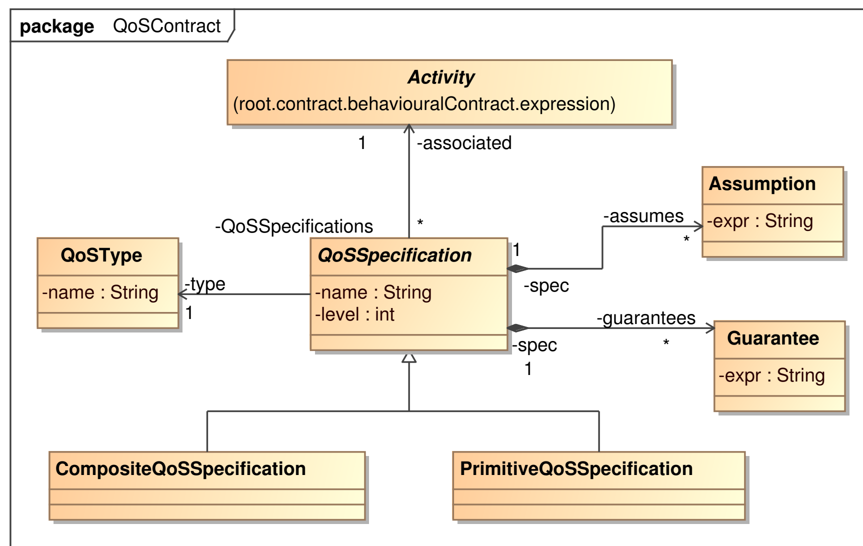

FIG. 5.13: Métamodèle des spécifications de QdS

Exemple. L'architecte a besoin d'exprimer le fait que le professionnel de santé doit recevoir les données médicales en moins de 10 secondes. Pour spécifier cette contrainte, il peut ajouter une spécification de QdS, nommé `QdS1`, en créant le modèle de spécification de QdS de la figure 5.14. La spécification `QdS1` est une hypothèse de type `MaximalResponseTime` et est associée à l'activité de réception de l'argument `data` par le point de communication `searchData` de l'entité `Client`. La valeur de la contrainte est la suivante : `T<10`

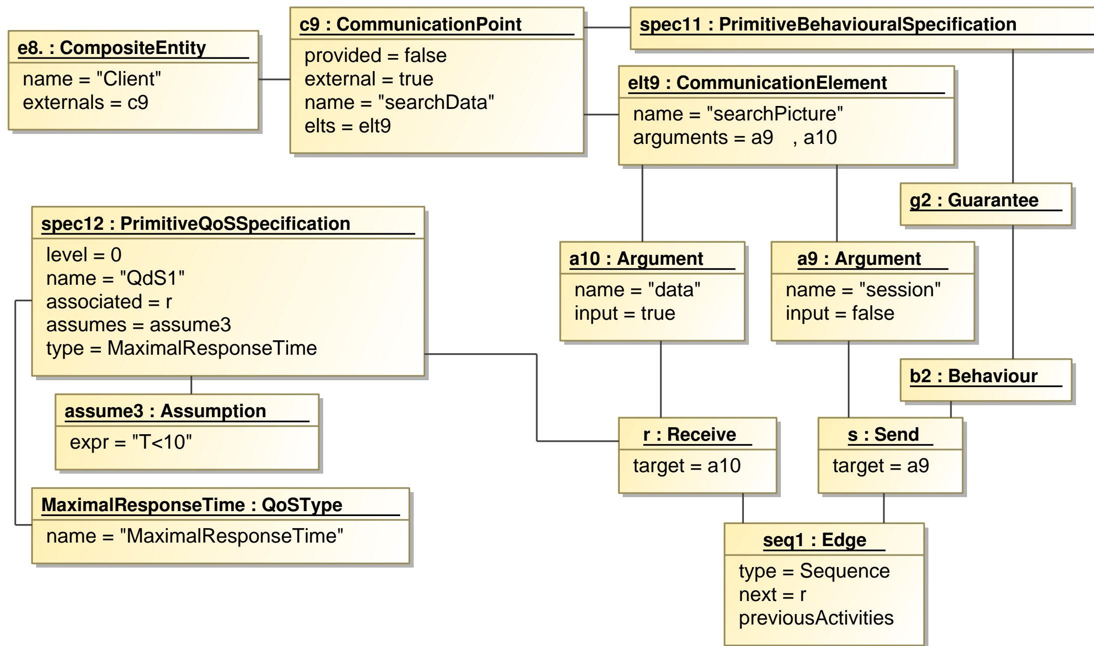

FIG. 5.14: Modèle de la spécification de QdS `QdS1`

5.3.6 Bilan

Pour concevoir un logiciel fiable, l'architecte a besoin de pouvoir spécifier les diverses propriétés applicatives de son logiciel. Or, très peu de modèles et de plates-formes offrent ce support. De plus, à notre connaissance, aucun modèle et aucune plate-forme ne permet la spécification des quatre catégories de propriétés applicatives. En effet, le support autorisant la spécification des propriétés applicatives est très disparates selon les modèles utilisés.

Notre objectif est donc de pallier ce manque. Ainsi, CALICO offre à l'architecte un moyen pour spécifier les exigences que le logiciel doit respecter, c'est-à-dire structurelle, comportementale, de flot de données et de QdS.

Pour permettre cette spécification, notre objectif n'est pas de redéfinir de nouveaux moyens de spécification, mais de permettre la réutilisation de la plupart des différentes approches existantes. Nous avons donc choisi d'avoir une approche extensible et générique, basée sur le paradigme hypothèse/garantie. De cette manière, l'intégration des approches basées sur ce paradigme est directe, ce qui correspond à la majorité des approches existantes.

5.4 Conclusion

Nous venons de présenter le mécanisme mis en place dans CALICO permettant à un architecte logiciel de concevoir et de faire évoluer la structure d'une architecture à composants ou orientées services et de spécifier les quatre catégories de propriétés applicatives, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. Notre approche est basée sur l'ingénierie dirigée par les modèles. CALICO offre un métamodèle générique pour décrire l'architecture et quatre métamodèles dédiés à l'expression des quatre catégories de propriétés applicatives.

Pour répondre à la problématique liée à la séparation des préoccupations, c'est-à-dire autoriser la conception de la structure d'un système indépendamment de toute plate-forme, nous avons élaboré un métamodèle de la structure du système générique. Ce métamodèle résulte d'une ana-

lyse de domaine de plusieurs modèles à composants et d'un modèle orienté services. Afin d'être le plus générique possible, il contient un minimum de concepts et de contraintes de règles de composition communs aux différents modèles. Ces règles portent sur la vérification de la stricte encapsulation des entités et sur l'unicité du nommage des éléments. Les règles de composition spécifiques à chaque plate-forme sont définies séparément du métamodèle grâce aux profils de plate-forme. Cette approche permet de vérifier si une architecture respecte les règles de composition de la plate-forme et peut donc en conséquence être déployée sur cette plate-forme. La migration vers une autre plate-forme est ainsi facilitée. L'architecte a besoin de changer le profil chargé et de mettre à jour les propriétés spécifiques à la plate-forme dans la mesure où la spécification de la structure n'est pas dépendante de la plate-forme.

Notre deuxième objectif concerne la spécification des propriétés applicatives. CALICO supporte la spécification des quatre catégories de propriétés applicatives. Ces spécifications reposent sur le paradigme hypothèse/garantie. De cette manière, CALICO est générique et extensible. En effet, CALICO autorise un expert du domaine à ajouter le support de nouvelles catégories de spécifications. Pour réaliser cette extension, l'expert de domaine a besoin de définir un nouveau métamodèle et de le coupler fortement avec le métamodèle de la structure du système. D'autre part, dans la mesure où le métamodèle de la structure est indépendant de toute plate-forme, la spécification des quatre niveaux des propriétés applicatives est possible pour les différentes plates-formes gérées par CALICO. Par exemple, un architecte peut spécifier les quatre niveaux de propriétés applicatives sur une architecture SCA, alors que nativement les plates-formes d'exécution de ce type ne le permettent pas.

Globalement, le mécanisme mis en place pour l'étape de conception permet une réunification générique des travaux existants dans le domaine de la conception. De plus, ces travaux qui étaient auparavant dédiés à une plate-forme sont maintenant rendus disponibles pour toutes les plates-formes prises en compte dans CALICO.

Chapitre 6

Étape d'analyse du système

Sommaire

6.1 Motivations **101**
6.1.1 Cas des interactions partiellement compatibles 102
6.1.2 Extensibilité 103
6.2 Couplage entre les analyses statiques et dynamiques **103**
6.2.1 Spécification des analyses dynamiques 103
6.2.2 Spécification des mécanismes d'observation 107
6.3 Analyse de la cohérence du système **110**
6.3.1 Composition des spécifications contractuelles 110
6.3.2 Contrat structurel 112
6.3.3 Contrat comportemental 113
6.3.4 Contrat de flot de données 115
6.3.5 Contrat de QdS 118
6.3.6 Intégration des outils d'analyse 121
6.4 Conclusion **123**

Une fois que l'architecte a conçu la structure du système et spécifié ses propriétés applicatives, la cohérence de l'architecture a besoin d'être analysée afin de vérifier que toutes les propriétés applicatives sont respectées. Ce chapitre présente les analyses statiques réalisées par l'outil d'analyse des interactions de CALICO lors de la seconde étape du cycle de développement. Plus précisément, la section 6.1 donne les motivations et les objectifs de l'étape d'analyse. La section 6.2 explique les mécanismes mis en œuvre dans CALICO pour traiter le couplage des analyses statiques et des analyses dynamiques. Finalement, la section 6.3 décrit les analyses structurelles, comportementales, de flot de données et de QdS.

6.1 Motivations

Après avoir conçu le système, il est extrêmement important de déterminer, le plus tôt possible dans le cycle de développement, si l'architecture conçue est cohérente. D'une manière générale, l'analyse de la cohérence correspond à vérifier la compatibilité des interactions entre les composants et les services du système, ainsi qu'à tester si toutes les propriétés applicatives du système sont satisfaites. Cette analyse ne doit pas se contenter d'analyser les composants ou les services qui sont directement en interaction, mais a besoin d'effectuer une vérification globale de toute l'architecture. Comme les systèmes logiciels sont complexes, l'architecte logiciel ne peut pas

prédire les propriétés de l'assemblage juste en composant manuellement les propriétés individuelles de chaque composant ou service. Donc, afin d'aider l'architecte à valider la cohérence de son architecture, les ADLs existants sont souvent associés avec des outils d'analyse, comme nous l'avons présenté dans la section 3.2 de l'état de l'art. Les travaux autour de SafArchie ont déduit que l'analyse statique d'une interaction conduit à trois types de résultats [Bar05] (cf. section 2.2.5 de l'état de l'art). L'interaction est incompatible si une des propriétés applicatives est violée. Elle est compatible si toutes les propriétés sont respectées. Elle est partiellement compatible quand l'analyse statique a besoin, pour être finalisée, d'informations qui ne peuvent pas être connues statiquement. L'analyse des interactions compatibles et incompatibles peut être résolue directement. Le cas des interactions partiellement compatibles n'est cependant pas aussi direct, comme l'explique la section suivante.

6.1.1 Cas des interactions partiellement compatibles

Dans le cas des interactions partiellement compatibles, l'architecte peut simplement considérer que son architecture est incohérente, comme le fait par exemple Wright [All97] ou Darwin [Mag99]. Cependant, cette solution peut produire des faux positifs, c'est-à-dire que l'architecture est déclarée incohérente alors qu'il est possible qu'aucune erreur ne se produise pendant l'exécution du système. Les autres solutions reposent sur une analyse dynamique de l'interaction. L'objectif de l'analyse dynamique est d'observer, pendant l'exécution du logiciel, si les données d'exécution respectent les propriétés applicatives du logiciel. Une possibilité de mettre en œuvre les analyses dynamiques est d'ajouter des assertions dans le code de l'application. Ces assertions sont exécutées lorsque les testeurs exécutent les scénarios d'utilisation du logiciel pendant l'étape de validation dynamique. Ainsi cette étape de validation dynamique permet de détecter les valeurs de données qui violent les propriétés applicatives.

Comme nous l'avons vu dans l'état de l'art, il existe plusieurs travaux, comme ConFract, qui reposent sur ce type de solution. Ces travaux proposent un moyen à l'architecte de spécifier ces assertions. Celles-ci sont ensuite automatiquement ajoutées dans le code du système. Cependant, ces assertions n'ont qu'une visibilité limitée et ne prennent pas en considération l'architecture globale de l'application. En effet, les données d'exécution sont analysées uniquement localement, sans prendre en compte les informations globales, comme par exemple le flot de contrôle. Cela limite donc la pertinence des résultats des analyses dynamiques.

D'autre part, peu de plates-formes proposent un mécanisme qui supporte l'analyse dynamique du système. En conséquence, le développeur n'a pas d'autre choix que de mettre lui même en œuvre ce mécanisme. Par exemple, une solution répandue consiste à ajouter manuellement le code des assertions dans l'implémentation du système. Il y a donc un enchevêtrement du code métier du système et du code spécifique aux analyses dynamiques. Cette solution a pour conséquence d'augmenter la complexité du code et de diminuer sa lisibilité. De plus, l'enchevêtrement du code des analyses dynamiques et du code métier accroît la complexité liée à la suppression du code des analyses dynamiques pour élaborer la version finale du logiciel.

Quelque soit l'approche existante utilisée, il y a un découplage entre les analyses statiques et dynamiques. En effet, les analyses dynamiques ne sont pas définies automatiquement en fonction du résultat des analyses statiques. C'est donc l'architecte qui doit comprendre la source du problème liée aux interactions partiellement compatibles afin de déterminer quelles analyses dynamiques il a besoin d'insérer, ainsi que leurs localisations. De plus, à cause de ce découplage, les analyses dynamiques produisent des résultats portant sur l'implémentation du système. Il est donc nécessaire que l'architecte logiciel établisse le lien causal entre les éléments d'implémentation et de conception pour identifier l'élément de conception qui est la source de l'erreur avant de pouvoir corriger sa conception.

Notre premier objectif est donc de prendre en charge des analyses statiques qui vont au delà des interactions compatibles et incompatibles. Les analyses doivent être capables de traiter aussi le cas des interactions partiellement compatibles. Cette prise en compte des interactions partiellement compatibles implique un couplage fort entre les analyses statiques et dynamiques. Il

faut faire en sorte que les analyses dynamiques requises soient générées directement à partir du résultat des analyses statiques sans intervention humaine. Pour réaliser ce couplage fort, nous souhaitons que notre solution soit indépendante des plates-formes et non invasive. Le lien doit être suffisamment clair et flexible pour être utilisable avec différentes plates-formes.

6.1.2 Extensibilité

L'analyse de la cohérence d'une architecture a besoin de prendre en considération les différentes catégories de propriétés applicatives du système, comme les propriétés structurelles, comportementales, de flot de données ou de QdS. Cependant, comme cela a été démontré dans l'état de l'art, bien qu'il existe un très grand nombre de travaux différents portant sur l'analyse de la cohérence, malheureusement, chacun de ces travaux se focalise uniquement sur une catégorie de propriété, comme la QdS. Or, afin d'augmenter le niveau de confiance que l'architecte peut avoir dans son système, il est nécessaire de prendre en compte les diverses propriétés applicatives du système. Cela implique donc de réutiliser ces différents travaux d'analyses.

Notre second objectif est donc de permettre l'intégration dans CALICO de la plupart des différents outils d'analyses existants. Pour réaliser cet objectif, les outils d'analyse existants ont besoin d'être découplés et externalisés de toute plate-forme d'exécution. CALICO doit donc proposer un cadre d'intégration, indépendant de toute plate-forme, guidant un expert du domaine lors de l'ajout d'une nouvelle analyse. De plus, l'expert doit pouvoir exprimer l'ordre de dépendance entre les différentes analyses. Globalement CALICO doit permettre la réutilisation des outils d'analyses afin de rendre possible leur exploitation indépendamment de la plate-forme cible.

6.2 Couplage entre les analyses statiques et dynamiques

Cette section décrit les mécanismes mis en œuvre dans CALICO pour traiter le couplage entre les analyses statiques et dynamiques. Ce mécanisme remplit deux objectifs. D'une part, il permet aux analyses statiques de générer, indépendamment de la plate-forme cible, les analyses dynamiques qui doivent être exécutées lors de l'étape de validation dynamique du logiciel par les testeurs. D'autre part, il autorise la description du mécanisme d'observation des données d'exécution requises par les analyses dynamiques, indépendamment de la plate-forme cible.

Concrètement, le couplage entre les analyses statiques et dynamiques repose sur deux métamodèles : le métamodèle de débogage et le métamodèle d'aspect. La section 6.2.1 présente le métamodèle de débogage qui permet aux analyses statiques de définir les analyses dynamiques requises. La section 6.2.2 détaille le métamodèle d'aspect qui autorise la description des mécanismes d'observation à mettre en place dans le logiciel, séparément des modèles de l'architecture. Ce métamodèle se base sur le paradigme de séparation des préoccupations.

6.2.1 Spécification des analyses dynamiques

D'une manière générale, une analyse dynamique peut être vue comme une action qui est exécutée pendant l'étape de validation dynamique du logiciel. Cette action peut être par exemple un test unitaire ou un test d'intégration (cf. section 2.2.5). Dans notre approche, le rôle des analyses dynamiques est de vérifier dynamiquement si les données d'exécution violent les propriétés applicatives du logiciel. Pour réaliser cela, nous mettons en œuvre les analyses dynamiques via l'exécution d'assertions.

Nous souhaitons donc que le métamodèle de débogage puisse permettre la spécification de ces analyses dynamiques à exécuter lors de la validation dynamique du logiciel par les testeurs. Ces analyses dynamiques ont besoin d'être réalisées lorsqu'une modification apparaît dans le

système en cours d'exécution. Il est donc nécessaire de spécifier le moment auquel l'analyse dynamique doit être exécutée. Il faut aussi pouvoir spécifier l'assertion de l'analyse dynamique, ainsi que l'action à faire lorsqu'une erreur est détectée.

Donc, toujours avec l'optique d'élaborer un canevas de développement le plus générique possible, nous avons décidé de réutiliser le mécanisme existant dans les applications autonomes pour concevoir notre métamodèle (cf. section 3.1.4). En effet, de manière générale, les applications autonomes s'adaptent automatiquement en fonction d'une modification dans le contexte d'exécution. Ces adaptations sont décrites à l'aide de règles d'adaptation. D'autre part, le support d'exécution des systèmes autonomes repose sur un mécanisme de boucle de contrôle autonome [KC03]. Cette boucle de contrôle est composée de quatre étapes. L'étape d'observation consiste à détecter toute modification dans le contexte d'exécution. Lors de cette étape, les modifications sont décrites sous forme d'événements. L'étape d'analyse correspond à analyser les événements afin d'identifier si une adaptation est nécessaire. L'étape de planification met en place les adaptations qui devront être exécutées en fonction des règles d'adaptation qui ont été définies. Finalement, l'étape d'exécution des adaptations applique les adaptations qui ont été planifiées.

Nous avons donc réutilisé ce mécanisme de boucle de contrôle pour définir le lien causal entre les analyses statiques et les analyses dynamiques. De cette manière, la spécification d'une analyse dynamique correspond à ajouter une règle d'adaptation. De ce fait, l'exécution des analyses dynamiques est automatiquement contrôlée par le mécanisme de boucle de contrôle.

A) Métamodèle de débogage

Nous avons défini un métamodèle de débogage afin de permettre aux outils d'analyses statiques de définir, indépendamment de la plate-forme d'exécution choisie par l'architecte, les analyses dynamiques qui ont besoin être exécutées. Ce métamodèle repose sur le paradigme événement/condition/action (ECA) [PSD98] qui est largement utilisé dans les systèmes autonomes. Comme le montre la figure 6.1, une analyse dynamique est définie à l'aide d'une règle (`Rule`). Une règle est composée de trois parties : un événement (`Event`), une condition (`Condition`) et une action (`Action`). L'événement représente une modification dans le contexte d'exécution, comme par exemple une donnée médicale qui entre dans le PDA. Il exprime le moment où l'analyse dynamique doit être exécutée. La condition est une expression booléenne portant sur l'événement. Elle correspond à l'assertion à tester. Par exemple elle peut tester si la taille de la donnée médicale dépasse les 10 mégaoctets. L'action décrit ce qu'il faut effectuer quand la condition devient vraie. Elle correspond à la finalisation de l'analyse de l'interaction partielle-

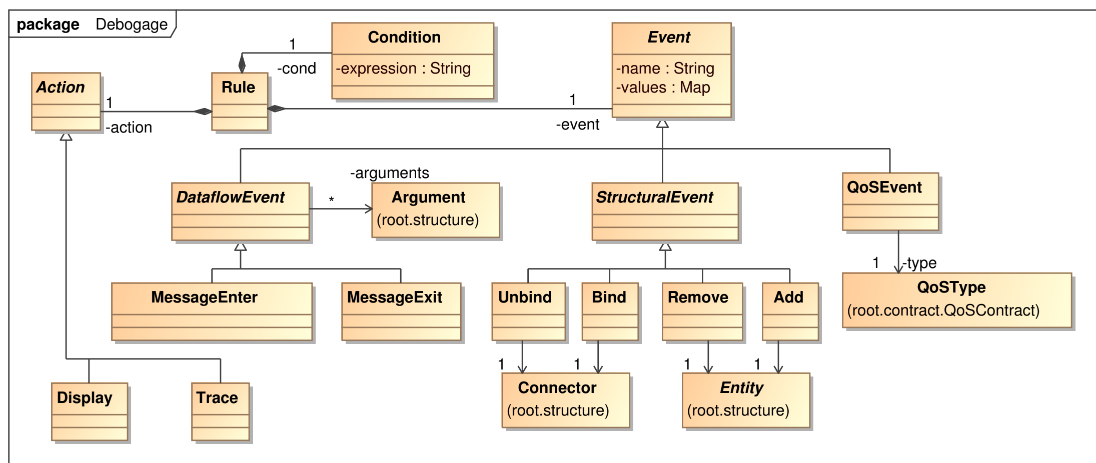

FIG. 6.1: Métamodèle de débogage

ment compatible. Par exemple, l'action peut notifier le problème à l'architecte si la finalisation de l'analyse produit une erreur.

Évènements. Afin de proposer une solution générique, nous avons effectué une analyse de domaine pour identifier les événements nécessaires pour effectuer une analyse dynamique. De plus, nous avons aussi besoin que ces événements soient communs aux différentes plates-formes à composants et aux différentes plates-formes orientées services. Nous avons trouvé trois catégories d'événements : les événements structurels, de flot de données et de QdS (cf. figure 6.1).

Les événements structurels expriment une modification de la structure du système. Il y a quatre types d'événements structurels. `Add` correspond à un ajout d'une entité dans le système, c'est-à-dire l'ajout d'un composant ou la publication d'un service dans un annuaire. `Remove` exprime la suppression d'une entité dans le système, c'est-à-dire la suppression d'un composant ou le déenregistrement d'un service dans un annuaire. `Bind` correspond à l'ajout d'un connecteur entre deux composants ou à la définition d'un `partnerLink` dans une architecture orientée services. `Unbind` correspond à la suppression d'un connecteur ou à la suppression d'un `partnerLink`.

Les événements de flot de données représentent la réification de la valeur des données échangées dans le système. Il y a deux types d'événements de flot de données. `MessageEnter` et `MessageExit` correspondent, respectivement, à la réception et à l'envoi d'un message par le point de communication d'un composant ou d'un service. Ces événements de flot de données englobent l'observation des propriétés comportemantales dans la mesure où le flot de contrôle de l'application peut-être déduit à partir de l'observation de tous les échanges de messages entre les composants/services.

Les événements de QdS représentent la modification de propriétés extra-fonctionnelles dans le contexte d'exécution. Il existe un type d'événement de QdS pour chaque type de QdS. Pour rappel, un type de QdS correspond à une catégorie de propriétés de QdS, comme la charge CPU en pourcentage ou la quantité de mémoire disponible en mégaoctet. Le lien entre un événement de QdS et un type de QdS est exprimé avec l'association entre la méta-classe `QoSEvent` et la méta-classe `QoSType` qui provient du métamodèle des spécifications de QdS de la figure 5.13.

Condition. La syntaxe de la condition est une expression booléenne très simple. Les variables peuvent être des entiers, des réels ou des chaînes de caractères. Les opérateurs de comparaison possibles sont $<, >, \leq, \geq, =, \neq$.

Action. L'action définit le type d'action à exécuter. Elle est indépendante de la plate-forme d'exécution choisie par l'architecte. La portée de l'action est volontairement limitée au niveau modèle, c'est-à-dire qu'une action a accès à tous les métamodèles de CALICO, mais ne peut jamais modifier directement l'architecture en cours d'exécution dans le niveau plate-forme. Cette approche garantit qu'aucune action ne pourra pas faire effondrer le système, parce que, comme nous l'avons vu dans l'aperçu de CALICO (cf. chapitre 4), les modifications effectuées dans les modèles sont propagées au système en cours d'exécution uniquement si aucune interaction incompatible n'est détectée.

Dans notre approche, les types d'actions sont dédiés à l'analyse dynamique du système. Nous proposons donc un mécanisme d'action extensible où un développeur de CALICO peut définir de nouveaux types d'actions. La version actuelle de CALICO est fournie avec deux actions prédéfinies (cf. figure 6.1). L'action `Display` est une action très élémentaire. Lorsque la condition devient vraie, elle affiche un message d'erreur à l'architecte. Elle lui indique la propriété applicative qui est violée afin que l'architecte corrige sa conception. L'action `Trace` permet à l'architecte logiciel de visualiser, au niveau modèle, le flot de messages échangés entre les composants et les services. De cette manière, l'architecte peut clairement visualiser le comportement réel de son système.

B) Discussion

Dans le cadre du livrable 2.3 du projet FAROS [DBFC+08] visant à définir les métamodèles de plates-formes, un métamodèle générique de plate-forme, visant à faire le lien entre le métier de l'application et la plate-forme d'exécution, a été conçu (cf. section 3.3). Ce métamodèle a pour objectif de permettre la spécification, l'observation et l'analyse dynamique de propriétés lors de l'exécution du système. Ce métamodèle possède une partie permettant de spécifier le moment de l'exécution de l'analyse dynamique. Cette partie contient plus de concepts que la partie événement de notre métamodèle de débogage.

Elle possède notamment les notions *avant* et *après* qui correspondent respectivement au moment juste avant et juste après que la modification du contexte se produise. Par exemple, il possède le concept `Bind` correspondant au moment juste avant la création d'un connecteur et le concept `AfterBind` correspondant au moment après la création d'un connecteur. Dans CALICO, notre objectif est de détecter, dans la mesure du possible, le problème avant que celui-ci ne provoque une erreur dans le système en cours d'exécution. En effet, notre but est d'intercepter la demande de modification qui se produit dans le système et de la réifier au niveau modèle. Cette demande de modification est ensuite contrôlée par notre mécanisme de boucle de contrôle. Donc dans notre approche, nos événements structuraux correspondent toujours à la notion *avant*. Par exemple, l'événement structurel `Bind` correspond au moment où un connecteur va être ajouté dans le système. Pour les événements de flot de données, nous avons décidé de ne pas être intrusifs, c'est-à-dire que nous observons uniquement les événements de flot de données visibles à partir de l'extérieur de l'entité. En conséquence, l'événement de flot de données `MessageEnter` correspond au moment où une donnée va entrer dans un point de communication et l'événement de flot de données `MessageExit` correspond au moment où une donnée est sortie d'un point de communication. Pour les événements de QdS, nous ne pouvons observer la modification des valeurs de propriétés extra-fonctionnelles que lorsqu'elles se sont produites. Donc ils correspondent au concept *après*.

D'autre part, le métamodèle de FAROS possède aussi des concepts concernant le démarrage et l'arrêt des entités et du système. Cependant ces concepts n'existent pas dans toutes les plates-formes. Dans CALICO, nous avons choisi de garder exclusivement les événements du cycle de vie communs à plusieurs plates-formes, c'est-à-dire que nous avons gardé l'intersection des différents événements. Au contraire dans FAROS, l'approche a été de faire la réunion de tous les événements de cycle de vie. Malheureusement, cette approche possède un inconvénient majeur qui va à l'encontre de nos objectifs, à savoir : pourvoir réutiliser les analyses indépendamment de la plate-forme cible. En effet, avec cette approche, suivant la plate-forme cible, il est nécessaire de restreindre les types d'événements autorisés car ils n'existent pas dans la plate-forme. Donc, puisque dans CALICO, une analyse statique est indépendante de la plate-forme, elle ne peut pas savoir quels sont les types d'événements qu'elle a le droit de manipuler.

De plus, dans CALICO, nous avons défini le type d'événement de QdS qui n'existe pas dans le métamodèle de FAROS. Ce type d'événement permet de représenter toute modification des valeurs de QdS dans le système, comme une augmentation de la charge CPU ou une diminution de la quantité mémoire disponible. Cela n'est pas possible avec le métamodèle FAROS, à moins d'utiliser l'événement de type applicatif, c'est-à-dire que l'observation des propriétés de QdS est à la charge de l'application. Cette solution allait à l'encontre de notre objectif, qui est de séparer clairement les préoccupations de validation du logiciel et les préoccupations métier.

Globalement, nous avons capturé dans le métamodèle de débogage tous les concepts, communs aux différentes plates-formes, qui sont suffisants pour faire l'analyse dynamique des quatre catégories de propriétés applicatives que nous avons identifiées. Ainsi, la généricité du métamodèle de débogage permet aux outils d'analyses statiques de spécifier les analyses dynamiques indépendamment de la plate-forme d'exécution cible. De plus, ces analyses dynamiques peuvent être réutilisées sur différentes plates-formes dans la mesure où la préoccupation de validation est clairement isolée du métier du système.

6.2.2 Spécification des mécanismes d'observation

Dans la section précédente, nous avons décrit comment les analyses statiques peuvent décrire les analyses dynamiques à exécuter lors de l'étape de validation dynamique du système. Cependant, afin de valider à l'exécution un système, les analyses dynamiques ont besoin d'observer les données d'exécution. Il est donc nécessaire de décrire le mécanisme d'observation des données d'exécution requises. De plus, nous désirons pouvoir clairement séparer la spécification du mécanisme d'observation et les spécifications portant sur le métier du système.

Toujours guidés par le même objectif de produire un canevas générique qui sépare bien les différentes préoccupations, nous avons décidé d'utiliser le paradigme de séparation des préoccupations en utilisant le développement par aspect (AOP pour "*Aspect Oriented Programming*") [KLM+97]. Dans notre approche, les préoccupations transverses correspondent aux préoccupations d'observation de données d'exécution. Notre métamodèle de structure du système est un métamodèle générique d'architecture à composants ou orientée services. Donc pour enrichir ce métamodèle avec les concepts d'aspect, nous avons besoin de réutiliser une approche qui unifie l'approche par composant et l'approche par aspect. Nous avons décidé de réutiliser les recherches de FAC (FAC pour "*Fractal Aspect Component*") [Pes07], mais d'autres travaux, tels que DAOP-ADL [PFT03], auraient aussi pu être utilisés.

A) Présentation de FAC

FAC est une extension du modèle à composants Fractal. Il ajoute sept concepts d'aspect au modèle Fractal, comme le montre la figure 6.2.

- Le *composant d'aspect* (`AspectComponent`) est un composant Fractal qui définit le comportement de l'aspect. Il joue le rôle de délégation entre un composant métier et le composant codant les fonctionnalités à tisser. Il possède une interface de conseil.
- L'*interface de conseil* (`AdviceInterface`) est une interface fournie d'un composant d'aspect. Elle définit le code qui doit être tissé avant, après ou autour d'une opération de composant interceptée par l'interface de tissage. Elle doit être de type `org.aopalliance.intercept.Interceptor`, qui est une interface de programmation communes pour différentes approches d'aspects dynamiques [1].
- L'*interface de tissage* (`WeavingInterface`) est une interface d'un composant aspectisable. Son rôle est d'intercepter les appels entrants et sortants du composant et de les rediriger vers le composant d'aspect cible.
- Un *composant aspectisable* (`AspectizableComponent`) est un composant qui supporte le tissage des préoccupations transverses. Il possède une interface de tissage.
- Une *liaison d'aspect* (`AspectBinding`) est une liaison transverse entre deux composants. Le composant client doit être un composant aspectisable et le composant serveur doit être un composant d'aspect.
- Le *domaine d'aspect* (`AspectDomain`) est un composant composite Fractal qui contient l'ensemble des composants connectés avec une liaison d'aspect. Il repose sur le support du partage de composants.

```
1 coupe      ::= <type> <composant>;<interface >;<methode>
2 type       ::= CLIENT | SERVER | BOTH
3 composant  ::= expression régulière sur le nom des composants
4 interface  ::= expression régulière sur le nom des interfaces
5 methode    ::= expression régulière sur le nom des opérations
```

LST. 6.1: Grammaire du langage de coupe de FAC

Dans FAC, il existe deux façons de tisser un composant d'aspect dans une architecture. La première façon consiste à ajouter manuellement une liaison d'aspect en appelant les APIs de

[1] http ://aopalliance.sourceforge.net

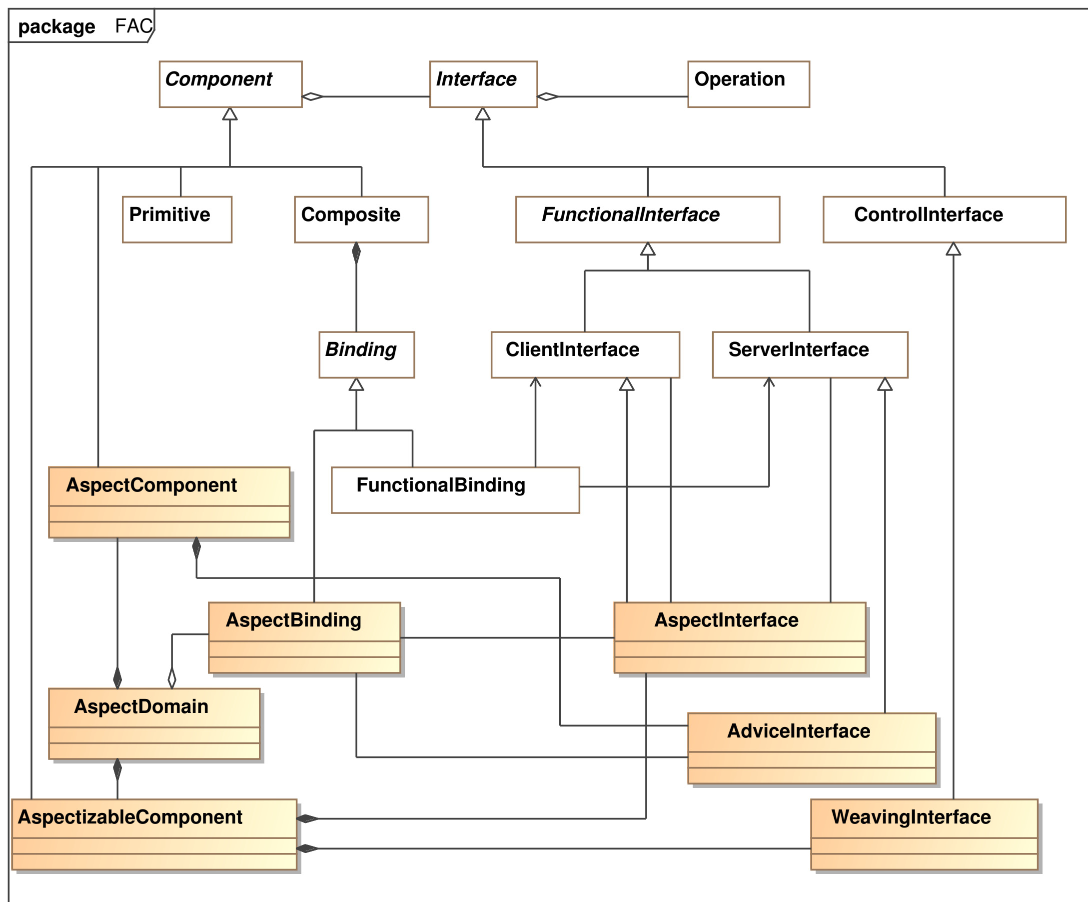

FIG. 6.2: Métamodèle de FAC

FAC. La seconde façon est d'écrire une expression de langage de coupe dont la grammaire est représentée dans le listing 6.1. L'expression de la coupe spécifie un type d'interception puis trois expressions régulières séparées par des points virgules. Ces expressions définissent le nom des composants, des interfaces et des opérations à intercepter.

B) Le métamodèle d'aspect de CALICO

Avec l'objectif de proposer un support générique des aspects, nous avons repris et adapté le métamodèle de FAC pour ajouter le support des aspects dans le métamodèle de structure du système de CALICO. Cependant, FAC est un modèle très proche du niveau implémentation. Il repose sur les fonctionnalités offertes par le modèle Fractal, comme le partage de composant ou les mécanismes d'interception. De plus, les informations de l'interface de conseil qui spécifie si le tissage doit être fait, avant, après ou autour d'une opération sont incluses dans le code du composant d'aspect.

Dans CALICO, nous désirons avoir un plus haut niveau d'abstraction et être indépendant des plates-formes d'exécution. Nous voulons que pour tisser un aspect dans la structure du système, il soit uniquement nécessaire de spécifier le minium de concepts, les autres concepts étant automatiquement déduits. De plus, la totalité des informations requises pour tisser les aspects doivent se trouver dans les modèles. Nous avons donc élaboré le métamodèle d'aspect de la figure 6.3.

Dans notre approche, la méta-classe `Aspect` joue le rôle de la liaison d'aspect. Elle connecte le point de communication fourni de l'entité codant la fonctionnalité transverse (as-

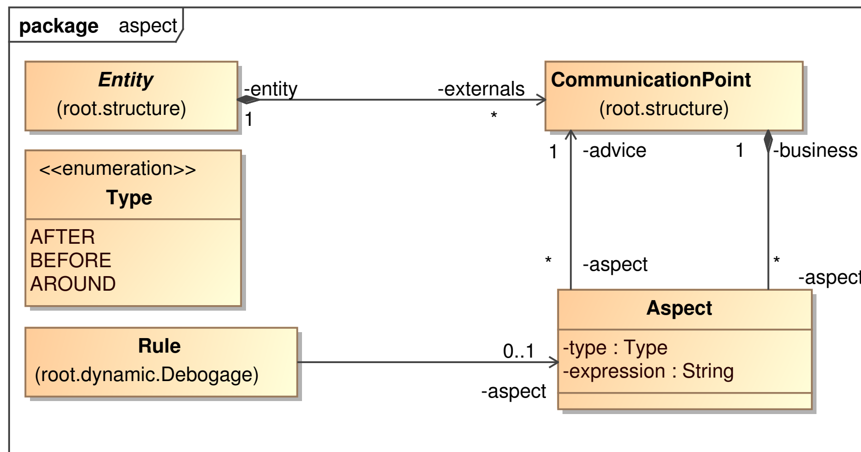

FIG. 6.3: Métamodèle d'aspect de CALICO

sociation `advice`) et le point de communication de l'entité métier (association `business`). Le point de communication de l'entité codant la fonctionnalité transverse doit être de type `org.aopalliance.intercept.Interceptor`. Le composant d'aspect n'est pas représenté dans notre métamodèle. En effet, il est automatiquement déduit à partir des informations stockées dans les attributs de la méta-classe `Aspect`. L'énumération `type` spécifie si l'aspect doit être tissé avant, après ou autour des éléments de communication. L'attribut `expression` est une chaîne de caractères correspondant à l'expression du point de coupe similaire à celle de FAC. La représentation concrète du composant d'aspect dans les plates-formes dépendra des fonctionnalités offertes par ces dernières. D'autre part, nous avons associé une analyse dynamique (`Rule`) à un aspect afin d'expliciter la relation entre l'analyse dynamique et le mécanisme d'observation. Ainsi, ce couplage permet de conserver les liens entre les analyses dynamiques et les mécanismes d'observation, ce qui rend possible la suppression des mécanismes d'observation lorsque ceux-ci ne sont plus utilisés par les analyses dynamiques.

Dans notre approche, nous n'avons pas les concepts de composants aspectisables et d'interfaces de tissage. Ces concepts permettent à un architecte de spécifier les composants pouvant être tissés. Dans CALICO, la préoccupation transverse correspond au mécanisme d'observation des données d'exécution du système et elle peut donc potentiellement concerner la totalité des entités du système. En conséquence, toutes les entités du système sont obligatoirement aspectisables.

Globalement, le métamodèle d'aspect de CALICO permet de spécifier le mécanisme d'observation des données d'exécution requises par les analyses dynamiques. De plus, ce métamodèle est très générique, il peut être utilisé dans un autre cadre que la spécification de fonctionnalité transverse d'observation de données. C'est pour cela que CALICO autorise l'architecte à manipuler ce métamodèle afin qu'il puisse exploiter le mécanisme d'aspect de CALICO pour spécifier ses propres fonctionnalités transverses.

C) Discussion

L'analyse dynamique des propriétés applicatives requiert l'observation de données d'exécution. Néanmoins, peu de plates-formes d'exécution permettent l'observation de données. Donc, pour spécifier la mise en œuvre de ce mécanisme, nous avons choisi de nous reposer sur l'AOP afin de séparer clairement les préoccupations métier des préoccupations d'observation.

Un des objectifs du projet FAROS est de mettre en œuvre la vérification des contrats dans les plates-formes d'exécution (cf. section 3.3). Ce projet a donc été aussi confronté à la problématique liée à l'observation des données d'exécution. Cette problématique a été prise en compte dans le livrable FAROS 2-3 qui décrit les métamodèles de plates-formes [DBFC+08]. Ce livrable contient

une partie qui concerne l'analyse dynamique des propriétés applicatives pendant l'exécution du système. Cette partie repose sur les notions d'`Observer`, de `Checker` et d'`Event`. Pour rappel, un `Event` représente une modification dans la plate-forme d'exécution, comme par exemple un message qui entre dans un composant. Un `Observer` est attaché à un `Event`. Son rôle est d'observer la modification dans la plate-forme qui correspond à l'`Event` auquel il est associé, puis de déclencher l'exécution du `Checker`. Le `Checker` est responsable d'effectuer l'analyse dynamique de la donnée d'exécution qui a été observée par l'`Observer`.

Notre métamodèle d'aspect est relativement proche du métamodèle de FAROS. Un `Observer` correspond au concept d'`Aspect`, c'est-à-dire un composant d'aspect. D'autres travaux, comme par exemple [OD07] et [BCC+09], proposent aussi des mécanismes qui reposent sur l'AOP pour traiter la problématique liée à l'observation de donnée d'exécution. Bien que notre approche soit similaire à ces travaux, notre différence provient de la manière dont CALICO propage les données d'exécution capturées. En effet, afin d'être le plus générique possible, les données capturées sont externalisées de la plate-forme. CALICO réifie automatiquement ces données d'exécution au niveau modèle en instanciant un événement `Event` conforme avec le métamodèle de débogage de la figure 6.1. De plus, nous avons aussi décidé d'externaliser le `Checker` de la plate-forme qui analyse, au niveau modèle, les événements (`Event`) émis par les observateurs intégrés dans l'application. Ainsi, en nous reposant sur le mécanisme de boucle de contrôle, nous apportons un support des analyses dynamiques indépendamment de la plate-forme cible, alors que dans FAROS il est nécessaire d'implémenter le `Checker` pour chacune des plates-formes cibles.

D'autre part, nous avons mis en évidence dans la section 3.1 de l'état de l'art que l'approche orientée aspect est un moyen approprié au niveau de la conception pour faire évoluer un logiciel. Nous avons donc décidé d'opter pour une approche AOP afin de permettre aussi à un architecte de décrire des préoccupations métiers transverses. Ainsi, le mécanisme d'aspect de CALICO peut être exploité, à la fois pour spécifier les informations transverses liées aux analyses dynamiques et à la fois par l'architecte.

6.3 Analyse de la cohérence du système

La section précédente a décrit les métamodèles de CALICO permettant de coupler les analyses statiques et dynamiques. Cette section présente comment l'outil d'analyse statique compose les spécifications d'hypothèse et de garantie des propriétés applicatives afin d'analyser la cohérence du système. Cette composition est générique et indépendante de tout modèle à composants et de tout modèle orienté services.

La section 6.3.1 explique l'algorithme général de composition des spécifications. Ensuite les sections 6.3.2 à 6.3.5 illustrent une concrétisation de l'algorithme général pour composer les spécifications structurelles, comportementales, de flot de données et de QdS. Les compositions sont expliquées à l'aide d'un exemple d'utilisation des métamodèles de débogage et d'aspect. Finalement la section 6.3.6 détaille le mécanisme mis en œuvre pour expliciter les dépendances entre les différentes analyses statiques.

6.3.1 Composition des spécifications contractuelles

Grâce à l'utilisation du paradigme hypothèse/garantie, il est possible de composer les spécifications contractuelles afin d'obtenir des contrats tels que $hypothèse(P) \rightarrow garantie(P)$, où l'opérateur $\rightarrow$ est une forme logique, qui exprime le fait que si l'hypothèse P est vraie alors la garantie P est vraie [AL93].

A) Algorithme de composition

Soit p un participant associé à la spécification contractuelle $Spec_p$. Nous définissons la spécification contractuelle $Spec_p$ d'un participant p par le couple (A_p, G_p) où A_p est l'hypothèse

de p et G_p est la garantie de p. Si un participant n'a pas d'hypothèse ou de garantie, ces hypothèses et garanties sont définies avec une valeur neutre par défaut. Le contrat $C_{\{p_1 \star p_2\}}$ provenant de l'interaction entre les participants p_1 et p_2 est un couple d'hypothèse et de garantie. Soit $a : C \rightarrow A$ une fonction de C vers A qui correspond à l'hypothèse du contrat C et soit $g : C \rightarrow G$ une fonction de C vers G qui correspond à la garantie du contrat C. Le contrat C est calculé tel que :

$$C_{\{p_1 \star p_2\}} = Spec_{p_1} \bullet Spec_{p_2}$$

avec $\bullet$ l'opérateur de composition des spécifications et $\star$ l'opérateur d'interaction entre les participants.

Les opérateurs $\bullet$ et $\star$ sont spécifiques à chaque catégorie de spécifications contractuelles, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. Le résultat du contrat est ensuite utilisé pour calculer incrémentalement la spécification $Spec_{\{p_1 \star p_2\}}$ de l'ensemble p_1 et p_2.

B) Définition des compatibilités des interactions

Soit le contrat $C_{\{p_1 \star p_2\}} = Spec_{p_1} \bullet Spec_{p_2}$ provenant de l'interaction entre p_1 et p_2. Nous définissons la compatibilité d'une interaction en nous basant sur le résultat du contrat, comme suit :

- L'interaction est compatible si $A_{P_2} \cap G_{P_1} = A_{P_2}$ et $A_{P_1} \cap G_{P_2} = A_{P_1}$.
- L'interaction est incompatible si $A_{P_2} \cap G_{P_1} = \emptyset$ ou $A_{P_1} \cap G_{P_2} = \emptyset$.
- L'interaction est partiellement compatible si ($A_{P_2} \cap G_{P_1} \neq A_{P_2}$ et $A_{P_2} \cap G_{P_1} \neq \emptyset$) ou ($A_{P_1} \cap G_{P_2} \neq A_{P_1}$ et $A_{P_1} \cap G_{P_2} \neq \emptyset$).

C) Projection de l'algorithme sur les métamodèles

Un analyse statique dans CALICO correspond à une transformation de modèle, comme le montre la figure 6.4. Elle prend en entrée trois modèles. Le modèle de structure du système contient la description structurelle de la totalité du système. Il permet à l'outil d'analyse statique de déterminer les entités qui interagissent entre elles. Ensuite, les modèles des spécifications contractuelles décrivent les spécifications d'hypothèses et de garanties qu'il faut valider. Finalement le dernier modèle est le modèle de mise à jour. Il représente les modifications structurelles

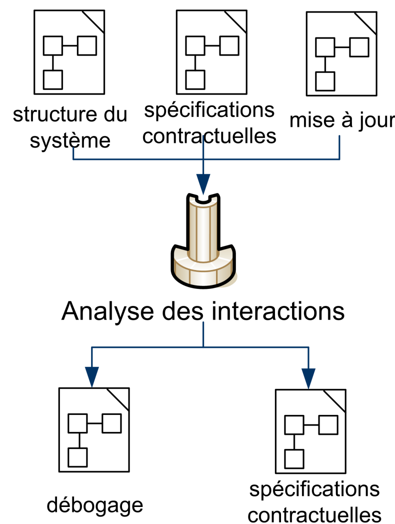

FIG. 6.4: Principe global d'une analyse statique

qui ont eu lieu depuis la dernière analyse statique. Ce modèle est détaillé dans la section 7.4 du chapitre 7. Ce modèle permet à l'outil d'analyse statique de valider le système de manière incrémentale en exploitant à la fois le résultat des analyses précédentes et en prenant en compte les modifications de la structure.

En sortie, l'analyse statique modifie les modèles de débogage et de spécifications contractuelles. Elle sauvegarde dans le modèle de spécification contractuelle le résultat de l'analyse statique. Pour chaque interaction partiellement compatible, l'analyse statique ajoute, en plus, dans le modèle de débogage une analyse dynamique qui sera exécutée pendant l'étape de validation dynamique du système. Pour chaque interaction incompatible, l'analyse statique notifie l'erreur à l'architecte.

Les sections suivantes illustrent l'utilisation du paradigme hypothèse/garantie pour composer les quatre catégories de spécification, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. Cependant, le support des analyses statiques de CALICO n'est pas limité à ces quatre exemples. CALICO est extensible et autorise l'intégration de nouvelles analyses statiques.

6.3.2 Contrat structurel

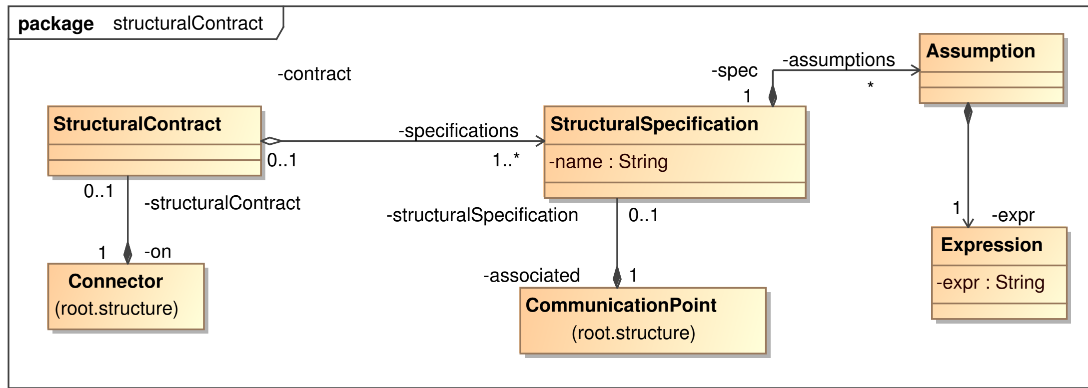

FIG. 6.5: Métamodèle des contrats structurels

Un contrat structurel résulte de l'interaction entre un ensemble de points de communication fournis P_1 et un ensemble de points de communication requis P_2 connectés ensemble via un connecteur, comme le montre le métamodèle des contrats structurels de la figure 6.5. Le contrat est calculé en composant les spécifications structurelles $Spec_{P_1}$ de P_1 et les spécifications structurelles $Spec_{P_2}$ de P_2 à l'aide de l'opérateur de composition and d'OCL (cf. tableau 6.1), tel que :

$$C_{\{P_1 \star P_2\}} = Spec_{P_1} \bullet Spec_{P_2} = \begin{cases} A = A_{P_1} \ and \ A_{P_2} \\ G = true \end{cases}$$

$C_{\{P_1 \star P_2\}}$	$a(C) =$	$g(C) =$
Connecteur ($P_1 \star P_2$)	$A_{P_1} \ and \ A_{P_2}$	$true$

TAB. 6.1: Opérateurs de composition des spécifications structurelles

Le contrat obtenu est associé au connecteur. Son expression correspond à un invariant OCL qui peut être vérifié statiquement par un validateur OCL.

Exemple. Pour illustrer la création de contrats structurels, nous reprenons l'exemple du DMP de la section 5.3.2. Nous analysons la cohérence de ce système en prenant en compte les spécifications structurelles. Dans le DMP, le point de communication `getTicket` de l'entité `SessionServer` porte une spécification structurelle $Spec_{SessionServer.getTicket} = (S1)$ (cf. section 5.3.2). Lorsque ce point de communication est connecté avec le point de communication `getTicket` de l'entité `Authentication`, un contrat structurel $C_{\{SessionServer.getTicket \star Authentication.getTicket\}}$ est calculé tel que :

$C_{\{SessionServer.getTicket \star Authentication.getTicket\}} = (A, true)$ où
$A = A_{SessionServer.getTicket}$ *and* $A_{Authentication.getTicket} =$
$'SessionServer.getTicket'.other.entity.name = "Authentication"$ *and true*

Ce contrat est valide dans l'architecture que l'on considère (cf. figure 5.3).

6.3.3 Contrat comportemental

Un contrat comportemental résulte de l'interaction entre un point de communication P_1 et un point de communication P_2 connectés ensemble avec un connecteur (cf. figure 6.6). Le contrat est attaché à ce connecteur. Il est calculé en composant les spécifications comportementales $Spec_{P_1}$ de P_1 et $Spec_{P_2}$ de P_2 avec le couple d'opérateurs $(\cup, \|)$ où $\cup$ correspond à la réunion des hypothèses et $\|$ est l'opérateur de synchronisation de SFSP (cf. tableau 6.2). Le contrat est donc calculé tel que :

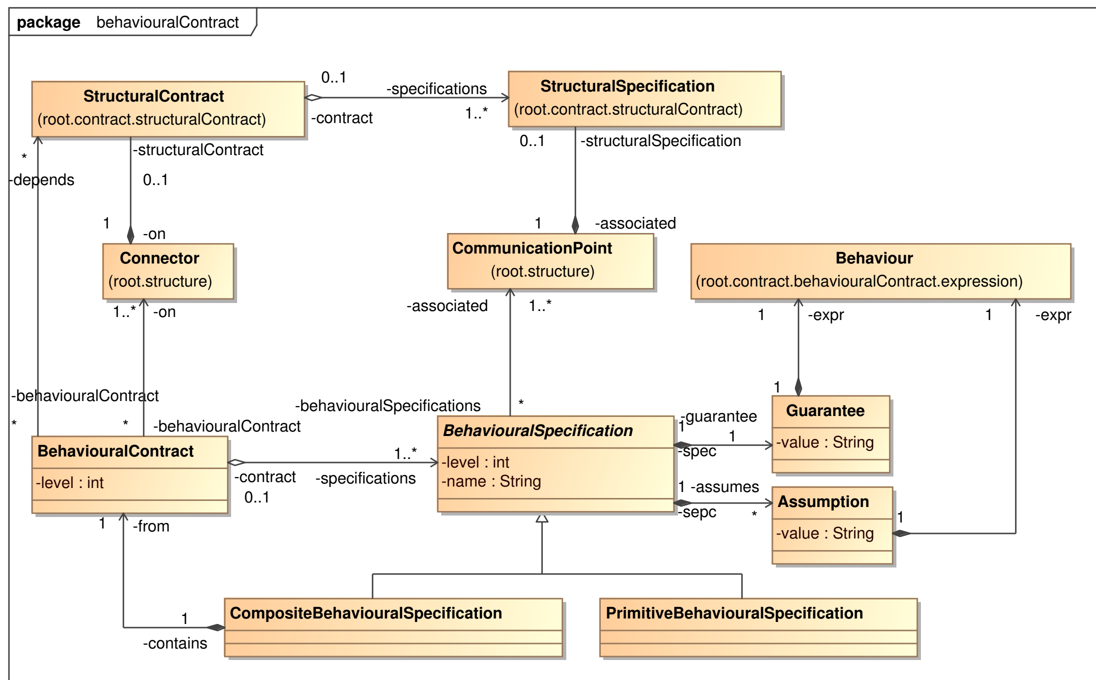

FIG. 6.6: Métamodèle des contrats comportementaux

$$C_{\{P_1 \star P_2\}} = Spec_{P_1} \bullet Spec_{P_2} = \begin{cases} A = A_{P_1} \cup A_{P_2} \\ G = G_{P_1} \parallel G_{P_2} \end{cases}$$

$C_{\{P_1 \star P_2\}}$	$a(C) =$	$g(C) =$
Connecteur ($P_1 \star P_2$)	$A_{P_1} \cup A_{P_2}$	$G_{P_1} \parallel G_{P_2}$

TAB. 6.2: Opérateurs de composition des spécifications comportementales

Le résultat du contrat exprime le flot de contrôle de l'assemblage $P = \{P_1 \star P_2\}$. L'algorithme de résolution est basé sur le mécanisme présent dans les algèbres de processus, comme CSP [Hoa85], FSP [Mag99] et SFSP [Bar05]. Cependant, notre graphe de flot de contrôle résultant de la composition est sensible au contexte. L'interaction est compatible si et seulement si le graphe de flot de contrôle de l'assemblage ne contient pas d'état puits, c'est-à-dire que le graphe ne contient aucun noeud s tel qu'il n'existe pas de chemin entre s et un noeud final. Cela correspond à la notion de compatibilité totale dans SFSP. L'interaction est incompatible quand aucun état final du graphe n'est accessible. L'interaction est partiellement compatible si le graphe contient au moins un noeud puits et au moins un noeud final accessible, c'est-à-dire qu'il existe un chemin entre un noeud initial et un noeud final.

Dans le cas des interactions partiellement compatibles, SFSP se contente de déclencher un avertissement à l'architecte. Dans CALICO, l'analyse statique ajoute une analyse dynamique dans l'architecture pour détecter le plus tôt possible l'interblocage. Cette analyse dynamique lève une erreur dès que le chemin inéluctable vers le puits est emprunté. Concrètement, l'analyse statique localise le noeud s le plus proche du noeud initial tel que tous les chemins issus de s mènent à un noeud puits. Pour ce noeud s, l'analyse statique ajoute une analyse dynamique telle que : l'événement représente un événement de flot de données, qui peut être un événement d'envoi ou de réception de messages selon que l'activité de s représente un état d'attente de réception de messages ou un état d'émission de messages. La condition est toujours vraie afin de capturer tous les messages qui empruntent le chemin inéluctable vers le noeud puits. L'action correspond à alerter l'architecte de l'interblocage. Elle montre aussi la spécification contractuelle qui a été violée et fournit la trace complète du message afin que l'architecte puisse identifier la source du problème.

Le résultat du contrat $C_{\{P_1 \star P_2\}}$ devient la spécification comportementale $Spec_{\{P_1 \star P_2\}}$ de l'assemblage $P = \{P_1 \star P_2\}$. Cette spécification est appelée spécification composite et est associée aux points de communication $P = \{P_1 \star P_2\}$.

Exemple. Nous reprenons l'exemple du DMP pour illustrer la composition des spécifications comportementales. Nous ne considérons que l'assemblage des entités `GlobalSearch` et `MedicalServer`. Lorsque le point de communication `getData` de l'entité `GlobalSearch` et le point de communication `getData` de l'entité `MedicalServer` sont connectés, les spécifications $Spec_{GlobalSearch.getData}$ et $Spec_{MedicalServer.getData}$ sont composées. Dans cet exemple, $Spec_{GlobalSearch.getData} = (true, B2)$ et $Spec_{MedicalServer.getData} = (B3, B1)$ (cf. section 5.3.3). Le contrat $C_{GlobalSearch.getData \star MedicalServer.getData}$ est calculé tel que :

$$C_{GlobalSearch.getData \star MedicalServer.getData} = \begin{cases} A = true \cup B3 \\ G = B2 \parallel B1 \end{cases}$$

Après la synchronisation de $B2$ et $B1$, la garantie correspond donc à :

```
?GlobalSearch.searchData.searchPicture.session ; !GlobalSearch.checkTicket.verifyTicket.ticket ;
?GlobalSearch.checkTicket.verifyTicket.result$ ; (!GlobalSearch.searchData.searchPicture.data$ |
!GlobalSearch.getData.getPicture.URL ; ?MedicalServer.getData.getPicture.URL ;
!MedicalServer.getData.getPicture.data$ ; ?GlobalSearch.getData.getPicture.data$ ;
!GlobalSearch.searchData.searchPicture.data$)
```

Et l'hypothèse est la suivante :

```
?SessionServer.checkTicket.verifyTicket.ticket;*;?MedicalServer.getData.getPicture.URL
```

Le graphe de flot de contrôle correspondant à la garantie ne possède pas d'interblocage. Il ne possède aucun noeud puits. Cependant, l'hypothèse n'est pas encore vérifiée car le point de communication `checkTicket` de l'entité `SessionServer` n'est pas appelé dans le flot de contrôle. Ce contrat sera donc valide quand l'entité `SessionServer` sera connectée à l'assemblage. Le contrat devient la spécification $Spec_{GlobalSerach \star MedicalServer}$ de l'assemblage entre `GlobalSearch` et `MedicalServer`.

Cette spécification va servir à calculer les spécifications des assemblages de manière incrémentale. De cette manière, la spécification comportementale $Spec_{Client \star GlobalSearch \star MedicalServer \star SessionServer}$ peut être calculée incrémentalement. La figure 6.7 représente graphiquement le graphe de flot de contrôle de cet assemblage. Pour des raisons de lisibilité, le modèle des contrats comportementaux n'est pas représenté.

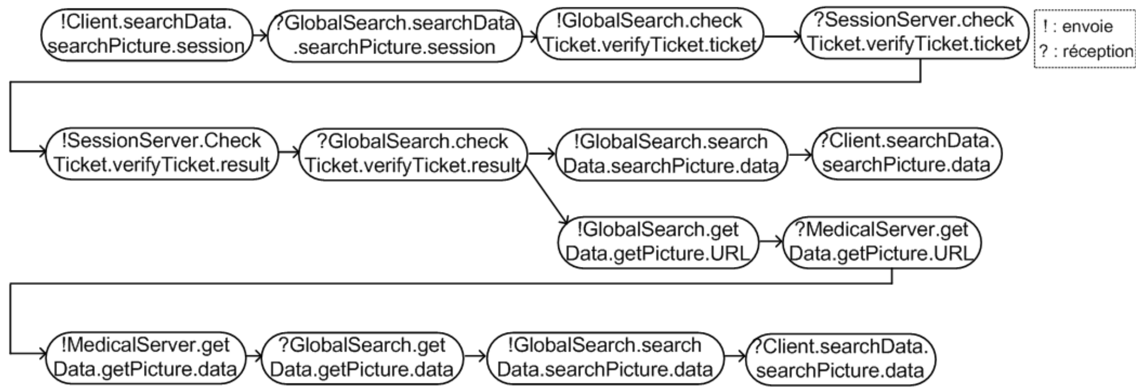

FIG. 6.7: Schéma du comportement de l'assemblage

6.3.4 Contrat de flot de données

L'analyse de flot de données repose sur le graphe de flot de contrôle calculé pendant l'analyse du comportement, comme le montre le métamodèle des contrats de flot de données de la figure 6.8. De manière générale, les spécifications de flot de données $Spec_{P_i}$ (`DataflowSpecification`) associées aux noeuds du graphe de flot de contrôle (`Activity`) sont propagées à travers le graphe de flot de contrôle. Cette propagation est similaire à l'algorithme de propagation de constante dans une analyse partielle de programme [Kil73]. Toutes les garanties, qui sont des postconditions, sont propagées en avant dans le graphe. Toutes les hypothèses, qui sont des préconditions, sont propagées en arrière dans le graphe jusqu'à la première alternative ou jusqu'au premier noeud dans le graphe qui transfère les données impliquées dans l'hypothèse. Ainsi, le contrat est composé en fonction de l'opérateur de composition du flot de contrôle, comme représenté dans le tableau 6.3.

Par exemple, si les spécifications $Spec_{P_1}$ et $Spec_{P_2}$ sont composées avec l'opérateur de divergence du flot de contrôle, alors le contrat $C_{P_1 \star P_2}$ est calculée comme cela est indiqué à la ligne 3 du tableau 6.3. L'hypothèse de $C_{P_1 \star P_2}$ correspond à l'ensemble de l'hypothèse de P_1 et de tous les hypothèses des contrats de chacune des branches divergentes i du flot de contrôle. En effet, pour exécuter une divergence parallèle du flot de contrôle, il faut pouvoir entrer dans toutes les branches divergentes i du flot. Donc il faut que les hypothèses A_{P_i} de chacune des branches soient validées. Cette analyse statique correspond à faire remonter dans le flot de contrôle les hypothèses. Notre analyse statique est sensible au contexte, donc il existe une garantie différente pour chacune des branches i du flot. Cette garantie correspond à l'intersection des garanties de

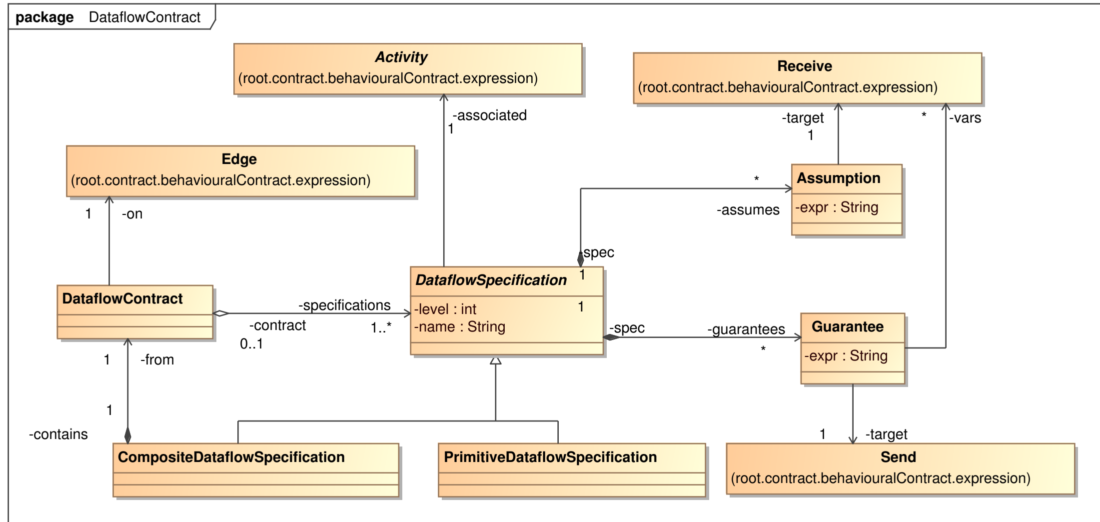

FIG. 6.8: Métamodèle des contrats de flot de données

$C_{P_1 \star P_2}$	$a(C) =$	$g(C) =$
Sequence ($P_1 \rightarrow P_2$)	$A_{P_1} \cap a(C_{P_1 \star P_2})$	$G_{P_1} \cap G_{P_2} \cap A_{P_2}$
P_1 Split P_i	$A_{P_1} \bigcap_i a(C_{P_1 \star P_i})$	$G_{P_1} \cap G_{P_i} \cap A_{P_i}$
P_i Join P_2	$A_{P_i} \cap a(C_{P_i \star P_2})$	$G_{P_i} \bigcap_i G_{P_2} \cap A_{P_2}$
P_1 Alternative P_i	$A_{P_1} \bigcup_i (a(C_{P_1 \star P_i}) \cap guard_i)$	$G_{P_1} \cap G_{P_i} \cap A_{P_i} \cap guard_i$

TAB. 6.3: Opérateurs de composition des spécifications de flot de données

P_i, de l'hypothèse de P_i et de la garantie de P_1.

Le résultat du contrat $C_{\{P_1 \star P_2\}}$ correspond à résoudre l'expression algébrique de l'hypothèse $a(C)$ en fonction de l'expression algébrique de la garantie $g(C)$:

$$C_{\{P_1 \star P_2\}} = resolve(a(C)) \text{ sachant } g(C)$$

Le résultat du contrat est une expression algébrique qui peut être soit $true$, soit $false$, soit une expression algébrique non résolue. L'interaction est compatible si le contrat est vrai. Elle est incompatible s'il est faux. Elle est partiellement compatible, si le contrat correspond à une expression algébrique non résolue.

Dans le cas où l'interaction est partiellement compatible, il est nécessaire d'analyser dynamiquement l'expression du contrat pendant l'exécution du système. Dans CALICO, nous désirons détecter le plus tôt possible le problème et éviter que le système entre dans la branche du flot de contrôle menant à la violation de la spécification. Pour cela, le contrat est propagé en arrière dans le flot de contrôle jusqu'à la première alternative ou jusqu'au dernier moment où les variables de l'expression du contrat sont impliquées dans l'activité du noeud du graphe. Soit s le noeud du graphe sur lequel le contrat est propagé. Dans le cas où des noeuds compris entre le noeud s et le noeud qui porte le contrat altèrent des variables utilisées dans le contrat, la manière dont ces variables sont modifiées a besoin d'être spécifiée dans les garanties de ces noeuds. Pour ce noeud s, l'analyse statique ajoute une analyse dynamique telle que : l'événement représente un événement de flot de données, qui peut être un événement d'envoi ou de réception de messages selon que s représente un état d'attente de messages ou un état d'émission de messages. La condition correspond à l'expression du contrat à résoudre. L'action correspond à alerter l'architecte de

la violation du contrat uniquement si la trace des messages échangés correspond au chemin dans le graphe de flot de contrôle conduisant au noeud s. Comme pour l'analyse du comportement, elle indique la spécification contractuelle qui a été violée et fournit la trace complète des messages échangés afin que l'architecte puisse identifier la source du problème.

Exemple. Cette section illustre la composition du flot de données avec l'exemple du système DMP. Dans cet exemple, nous considérons la composition des entités `Client`, `GlobalSearch`, `MedicalServer` et `SessionServer`. La composition des spécifications de flot de données repose sur le graphe de flot de contrôle de cet assemblage, représenté à la figure 6.7. Les noeuds de ce graphe sont associés avec les spécifications de flot de données `D1` à `D4`, comme cela est représenté sur la figure 6.9 (cf section 5.3.4).

En appliquant, l'algorithme de composition, les hypothèses sont propagées en arrière dans le flot et les garanties sont propagées en avant. Par exemple, la garantie `D3`, spécifiant que l'entité `GlobalSearch` ne modifie pas les données et la garantie `D2` qui exprime que les données médicales sortant de l'entité `MedicalServer` ont toujours une taille inférieure à 20 gigaoctets, sont propagées en avant dans le flot de contrôle jusqu'à l'activité de réception d'une donnée par le client. L'hypothèse `D1` de l'entité `Client`, qui spécifie que le client ne doit pas recevoir de donnée de taille supérieure à 10 mégaoctets, est propagé en arrière jusqu'à l'activité d'envoi d'une donnée par le serveur.

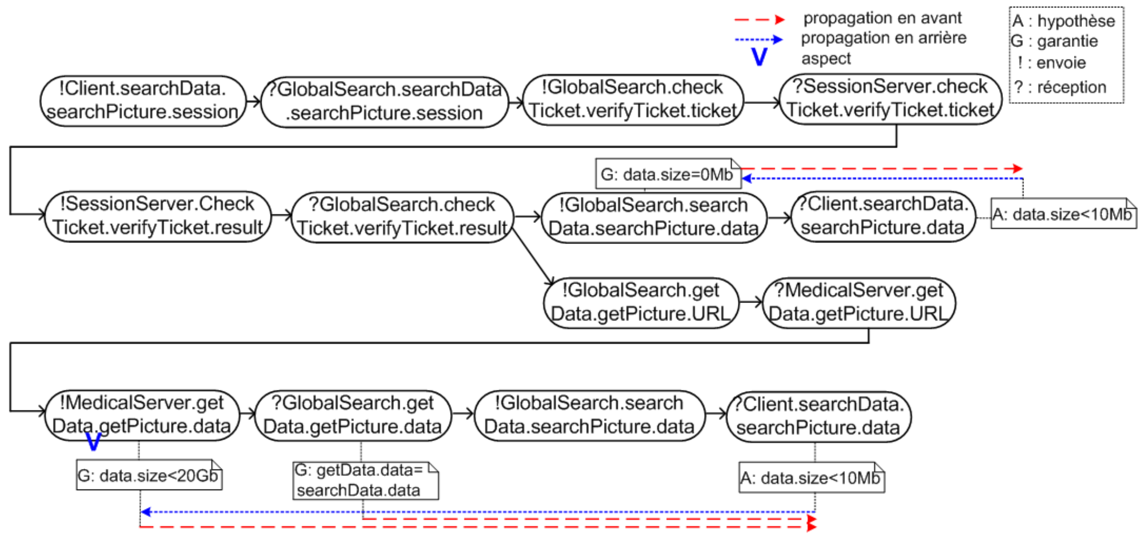

FIG. 6.9: Schéma de composition des spécifications de flot de données dans l'assemblage

Donc le contrat C qui est attaché à l'activité d'envoi d'une donnée par le serveur est :

$$C = \begin{cases} A = D1 \\ G = D2 \text{ et } D3 \end{cases}$$

$$C = \begin{cases} A = searchData.searchPicture.data.size < 10Mb \\ G = getData.getPicture.data.size < 20Gb \text{ et } searchData.searchPicture.data = getData.getPicture.data \end{cases}$$

Il est calculé localement :

$C = resolve(searchData.searchPicture.data.size < 10Mb)$ sachant
$getData.getPicture.data.size < 20Gb$ et $searchData.searchPicture.data = getData.getPicture.data$

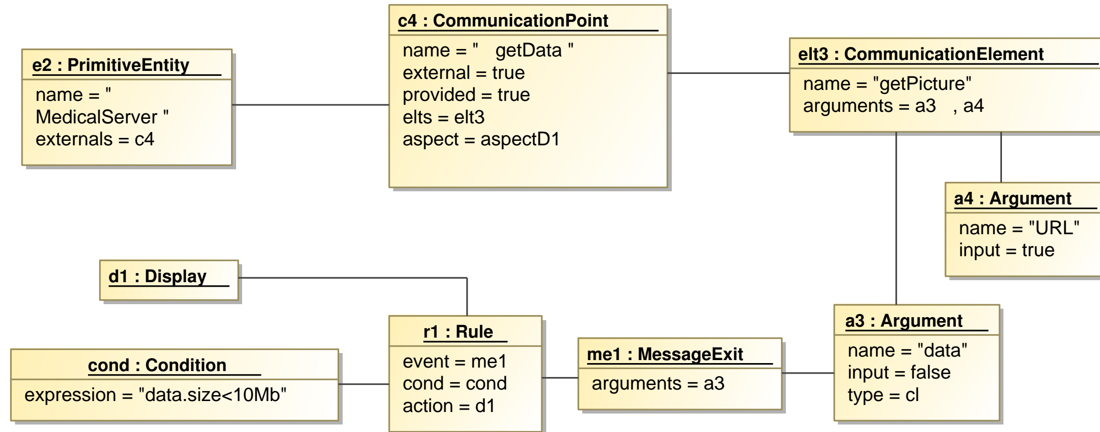

FIG. 6.10: Analyse dynamique de la spécification de flot de données `D1`

Après simplification automatique de l'équation grâce à un outil de calcul symbolique, le contrat à résoudre est :

$C = resolve(data.size < 10Mb)$ sachant $data.size < 20Gb$

Dans ce cas, la garantie n'est pas suffisante pour valider statiquement le contrat. L'interaction entre ces entités est donc partiellement compatible. L'expression du contrat est donc propagée en arrière jusqu'à l'activité d'envoi d'une donnée par l'entité `MedicalServer`. Afin de vérifier dynamiquement l'interaction, l'outil d'analyse statique ajoute une analyse dynamique dans le modèle de débogage, comme le montre le modèle de la figure 6.10. Le rôle de l'analyse dynamique est de résoudre le contrat C lors de l'exécution du logiciel. L'analyse dynamique est créée telle que : l'événement correspond à l'événement d'envoi d'une donnée par le point de communication `getData` de l'entité `MedicalServer`. La condition est l'expression : $data.size < 10Mb$. Le rôle de l'action sera de tester si la trace des messages échangés correspond bien au deuxième chemin du flot de contrôle, représenté en bas dans la figure 6.9. Si la trace correspond, l'action notifiera la violation de la spécification de flot de données `D1` à l'architecte.

Les spécifications de flot de données sont composées de la même manière dans la seconde branche de l'alternative du flot de contrôle. Dans cette branche, la garantie `D4` spécifie que la taille de la donnée est nulle. Donc, après la propagation de la garantie, le contrat suivant doit être résolu :

$C = resolve(searchData.data.size < 10Mb)$ sachant $searchData.data.size = 0Mb$

Ce contrat est toujours vrai. Donc l'interaction est compatible et aucune analyse dynamique n'est nécessaire.

6.3.5 Contrat de QdS

Un contrat de QdS $C_{\{P_1 \star P_2\}}$ résulte de la composition de spécifications de QdS $Spec_{P_1}$ et $Spec_{P_2}$ de même type, attachées au flot de contrôle, comme le montre la figure 6.11. Les contrats de QdS sont calculés en composant incrémentalement les spécifications de QdS. L'opérateur de composition des spécifications de QdS dépend, à la fois de l'opérateur de composition du flot de contrôle et du type de QdS. Donc, nous avons défini un métamodèle de type de QdS qui permet, à un expert du domaine, d'associer à chaque type de QdS (`QoSType`), l'opérateur de composition `QoSCompositionOperator` en fonction de l'opérateur de composition du flot de

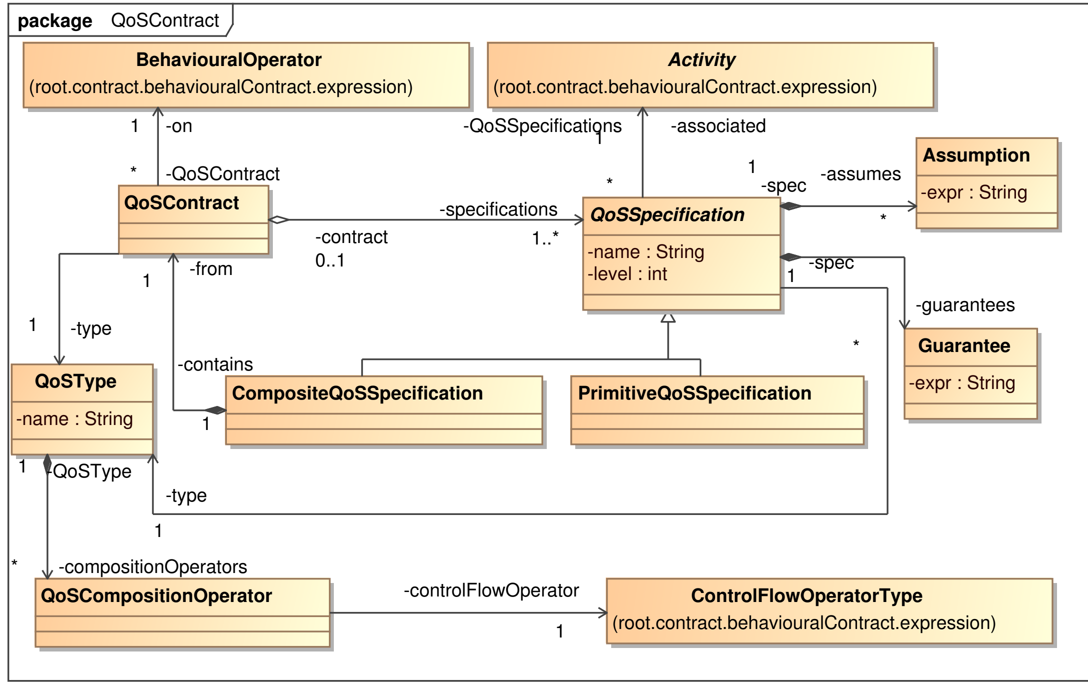

FIG. 6.11: Métamodèle des contrats de QdS

$C_{P_1 \star P_2}$	$a(C) =$	$g(C) =$
Sequence ($P_1 \to P_2$)	$sum(A_{P_1}, A_{P_2})$	$sum(G_{P_1}, G_{P_2})$
P_1 Split P_i	$max(A_{P_1}, A_{P_2})$	$max(G_{P_1}, G_{P_2})$
P_i Join P_2	$max(A_{P_1}, A_{P_2})$	$max(G_{P_1}, G_{P_2})$
P_1 Alternative P_i	$max(A_{P_1}, A_{P_2})$	$max(G_{P_1}, G_{P_2})$

TAB. 6.4: Exemple d'opérateurs de composition de la QdS (temps de réponse maximal)

$C_{P_1 \star P_2}$	$a(C) =$	$g(C) =$
Sequence ($P_1 \to P_2$)	$sum(A_{P_1}, A_{P_2})$	$sum(G_{P_1}, G_{P_2})$
P_1 Split P_i	$sum(A_{P_1}, A_{P_2})$	$sum(G_{P_1}, G_{P_2})$
P_i Join P_2	$sum(A_{P_1}, A_{P_2})$	$sum(G_{P_1}, G_{P_2})$
P_1 Alternative P_i	$min(A_{P_1}, A_{P_2})$	$min(G_{P_1}, G_{P_2})$

TAB. 6.5: Exemple d'opérateurs de composition de la QdS (temps de réponse minimal)

contrôle `ControleFlowOperatorType`, c'est-à-dire la séquence, l'alternative, la fusion et la séparation du flot de contrôle. Par exemple, [XCHZ07] a défini les opérateurs de composition de la QdS pour le temps de réponse maximal (cf. tableau 6.4) et minimal (cf. tableau 6.5). Par exemple, la tableau 6.4 correspond au type de QdS : "temps de réponse maximal". Le temps de réponse maximal théorique d'une exécution en parallèle des différentes branches T_i (lors d'une fusion ou d'une séparation du flot de contrôle) est égal au temps de réponse maximal entre les branches T_i, c'est-à-dire $\underset{i}{MAX}(T_i)$.

Globalement, le concept de type de QdS permet à un expert du domaine de définir les opérateurs de composition de la QdS séparément de l'algorithme qui parcourt le flot de contrôle. Cela facilite donc l'intégration dans CALICO de nouvelles analyses de QdS.

La validation du contrat de QdS $C_{\{P_1 \star P_2\}}$ de type T correspond à résoudre l'équation algébrique de l'hypothèse $a(C)$, sachant que la garantie $g(C)$ est vraie.

$$C_{\{P_1 \star P_2\}} = resolve(a(C))$$
$$\text{sachant que } g(C) = true$$

Le résultat du contrat de type T est une équation algébrique qui peut être soit $true$, soit $false$, soit non résolue. L'interaction est compatible si l'équation est vraie. Elle est incompatible si l'équation est fausse. Elle est partiellement compatible, si l'équation ne peut pas être résolue statiquement.

Dans le cas où l'interaction est partiellement compatible, il est nécessaire de finir la résolution de l'équation pendant l'exécution du système. Soit a l'arc qui porte le contrat. Soit le noeud s qui est l'origine de l'arc a. L'outil d'analyse statique ajoute pour ce noeud s une analyse dynamique dans le modèle de débogage telle que : l'événement représente un événement de QdS de type T, c'est-à-dire du type du contrat à valider. La condition correspond à l'expression du contrat à tester. L'action correspond à alerter l'architecte de la violation du contrat uniquement si la trace des messages échangés correspond au chemin dans le graphe de flot de contrôle conduisant au noeud s. Elle indique aussi la spécification contractuelle qui a été violée.

Exemple. Cette section illustre comment les spécifications de QdS sont composées dans le système DMP. La connexion entre l'entité `GlobalSearch` et `SessionServer` engendre la composition en séquence des spécifications de QdS telle que :

$$Spec_{GlobalSearch \star SessionServer} = Spec_\alpha \begin{cases} A = A_{GlobalSearch} + A_{SessionServer} \\ G = G_{GlobalSearch} + G_{SessionServer} \end{cases}$$

La connexion entre l'entité `GlobalSearch` et `MedicalServer` engendre la composition incrémentale des spécifications de $Spec_{GlobalSearch \star SessionServer} = Spec_\alpha$ et de $Spec_{MedicalServer}$. Dans cet exemple, l'opérateur de flot de contrôle est une alternative, alors la composition correspond à :

$$Spec_{\alpha \star MedicalServer} = \begin{cases} A = A_\alpha + A_{MedicalServer} \text{ si } checkTicket.result = true,\ A_\alpha \text{ sinon} \\ G = G_\alpha + G_{MedicalServer} \text{ si } checkTicket.result = true,\ A_\alpha \text{ sinon} \end{cases}$$

Si l'architecte garantit que le temps de réponse maximal de l'entité `MedicalServer` est de 6 secondes, que le temps de réponse maximal de l'entité `GlobalSearch` est de 1 seconde et que celui de l'entité `SessionServer` est de 2 secondes, alors le temps de réponse maximal de l'assemblage est de : $1 + 2 + 6 = 9$ si $result = true$, $1 + 2 = 3$ si $result = false$.

Dans cet exemple, le système DMP contient une spécification de QdS `QdS1` de type "temps de réponse maximal", qui est attachée au point de communication `getData` de l'entité `Client`. Cette spécification exige que le temps de réponse soit inférieur à 10 secondes (cf. section 5.3.5). Dans ce cas, le contrat est valide.

Cependant, dans la majorité des cas, il est difficile de pouvoir spécifier des garanties sur le temps de réponse. Donc, par manque d'informations statiques, l'interaction n'est que partiellement compatible. En conséquence, l'analyse statique a besoin d'ajouter une analyse dynamique. Elle définit donc une analyse dynamique dans le modèle de débogage telle que : l'événement est un événement `QoSEvent` de type `MaximalResponseTime`, la condition est $T < 10$ et l'action consiste à signaler la violation de la propriété `QdS1` à l'architecte.

6.3.6 Intégration des outils d'analyse

Ce chapitre est une discussion sur les mécanismes de CALICO permettant l'intégration uniforme des outils d'analyse. La première section explique comment CALICO prend en compte les dépendances entre les analyses. La seconde section discute sur les limitations de la composition incrémentale. Finalement, la troisième section présente les limitations de l'approche.

A) Ordre des analyses

Les contrats doivent être composés dans un ordre précis, car certaines analyses statiques peuvent reposer sur le résultat d'autres analyses statiques. Par exemple, l'analyse de flot de données se base sur le graphe de flot de contrôle produit par l'analyse du flot de contrôle.

Afin de permettre l'expression de ces dépendances, CALICO dispose du métamodèle d'intégration dont un sous ensemble est représenté sur la figure 6.12. Ce métamodèle autorise un expert du domaine à intégrer des nouvelles analyses statiques dans CALICO. Concrètement, l'expert définit l'outil d'analyse statique `Analysis` qu'il inclut dans CALICO, décrit par la métaclasse `CALICO`. Un outil d'analyse statique possède un nom et une classe d'implémentation qui correspond à l'exécutable à lancer par CALICO pour effectuer l'analyse. L'expert peut aussi spécifier les dépendances avec les autres outils grâce à l'association `dependencies`. Dans le cas où une analyse statique est incompatible avec une autre analyse statique, il peut aussi exprimer des conflits à l'aide de l'association `conflicts`.

Tous les outils des analyses statiques prises en compte dans CALICO sont référencés par l'association `analysis` de telle sorte qu'un architecte logiciel peut sélectionner les analyses qu'il désire en parcourant le contenu de cette classe et en les référençant à l'aide de l'association `selectedAnalysis`. De plus, afin de garantir que les outils d'analyse statique requis par les analyses désirées sont aussi sélectionnés, des contraintes OCL sont ajoutées dans le métamodèle d'intégration. Ces contraintes vérifient aussi qu'il n'y a pas de conflit entre les différentes analyses sélectionnées par l'architecte (cf. listing 6.2).

Ce métamodèle d'intégration est aussi utilisé par CALICO pour exécuter automatiquement les analyses statiques en respectant l'ordre. Cependant ces analyses statiques sont toutes exécutées en séquence, même si elles n'ont aucune relation de dépendance entre elles. Nous avons choisi explicitement de forcer un ordre strict et de ne pas les exécuter en parallèle afin d'éviter tout problème lié à une modification concurrente des modèles.

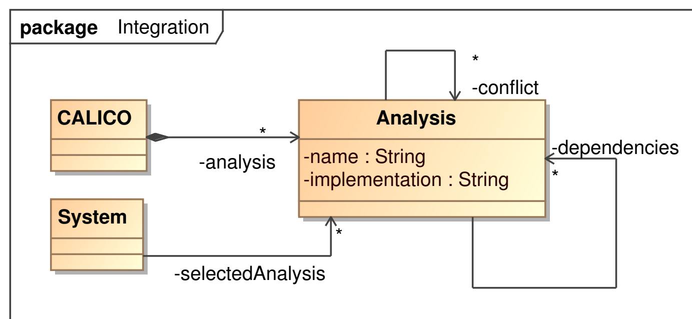

FIG. 6.12: Métamodèle d'intégration des outils d'analyse

```
1 context System
2   inv: self.selectedAnalysis->forAll(a:Analysis | self.selectedAnalysis->include(a.dependencies))
3   inv: self.selectedAnalysis->forAll(a:Analysis | not self.selectedAnalysis->include(a.conflict))
```

LST. 6.2: Contraintes OCL du métamodèle d'intégration

B) Composition incrémentale

La très grande majorité des outils d'analyse statique vérifie toujours la totalité des interactions du système. Cependant, CALICO est un canevas de développement incrémental et itératif. Donc CALICO permet une conception incrémentale du système, c'est-à-dire que lors des itérations suivantes du cycle de développement, l'architecte modifie quelques éléments de son architecture. Donc, au final seules quelques interactions du système ont été modifiées et ont donc besoin d'être revalidées. Afin de permettre cette validation incrémentale, CALICO garde dans les modèles les résultats des analyses statiques précédentes et les fournit en entrée aux outils d'analyse. De plus, les outils d'analyse statique peuvent exploiter le modèle de mise à jour, qui décrit l'ensemble des modifications structurelles qui ont eu lieu dans l'architecture depuis la dernière analyse, afin de détecter les interactions qui doivent être revalidées. Par exemple, l'outil d'analyse de SFSP effectue une vérification incrémentale du flot de contrôle [Bar05]. Avec ce principe d'analyse, les spécifications $Spec_{P_1 \cup P_2}$ forment un arbre de composition. Les feuilles de l'arbre correspondent aux spécifications primitives écrites par l'architecte et les noeuds de l'arbre correspondent aux spécifications composites portant sur un sous ensemble du système. De cette manière, la racine de l'arbre est la spécification de l'ensemble de système.

D'autre part, peu d'outils d'analyse statique ont été conçus pour effectuer des vérifications incrémentales. Donc, afin d'être compatible avec ces outils, CALICO offre aussi aux outils la possibilité de revalider la totalité des interactions, bien que cette méthode ait un impact sur le temps d'analyse.

C) Limitations

Afin de pouvoir intégrer le maximum d'outils d'analyse statique existants, indépendamment de la plate-forme d'exécution, nous avons effectué une analyse de domaine dans la section 3.2 de l'état de l'art. L'objectif de l'analyse de domaine est de déterminer, pour chaque catégorie de propriétés applicatives, les informations qui sont requises par les outils d'analyse existants pour valider une architecture logicielle. Nous en avons déduit que les outils existants ont besoin de connaître la structure de l'architecture logicielle et les propriétés applicatives à vérifier. De plus, dans le cas des analyses incrémentales, les outils ont aussi besoin des informations concernant l'évolution de l'architecture.

Nous avons donc élaboré le cadre d'intégration des analyses de CALICO de telle sorte qu'il donne aux analyses statiques l'accès au métamodèle de structure du système, aux métamodèles des contrats et au métamodèle de mise à jour. Le métamodèle de structure du système et les métamodèles des contrats contiennent toutes les informations liées à la structure et aux propriétés applicatives requises par les outils. Le métamodèle de mise à jour décrit les modifications structurelles qui sont nécessaires pour effectuer les analyses incrémentales. Ce métamodèle sera détaillé dans le chapitre 7. En conséquence, seules les analyses qui ont besoin d'autres informations, comme par exemple les analyses portant sur des concepts spécifiques à une plate-forme, ne sont pas prises en compte dans notre approche. Néanmoins ce type d'analyse ne représente qu'une minorité des analyses existantes. Notre approche autorise donc l'intégration de la plupart des outils d'analyse.

Globalement, CALICO a été construit pour permettre la réutilisation de la plupart des outils d'analyse statique existants, comme par exemple OCL, CSP, FSP. Pour réaliser cet objectif, nous avons externalisé le support des analyses des plates-formes d'exécution. De plus, ce support repose sur le paradigme hypothèse/garantie, ce qui permet une intégration directe des outils d'analyse qui se basent sur ce paradigme. Néanmoins, CALICO ne traite pas la problématique liée à l'interopérabilité entre différents outils portant sur la même catégorie de propriétés applicatives. Par exemple, CALICO ne permet pas de composer une spécification CSP avec une spécification FSP. Donc afin d'éviter un conflit, CALICO permet de déclarer ces analyses incompatibles. Ainsi, pour chaque catégorie de propriétés applicatives, au plus seule une analyse statique peut être activée.

6.4 Conclusion

Nous avons présenté l'outil d'analyse des interactions de CALICO qui permet d'analyser la cohérence de l'architecture. Cet outil supporte la validation statique du système de manière uniforme et itérative. Notre approche supporte l'analyse des quatre catégories de spécifications, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. Les spécifications reposent sur le paradigme hypothèse/garantie. Cela permet donc la composition incrémentale des spécifications en prenant en compte les évolutions produites par l'architecte lors de chaque itération du cycle de développement.

Notre approche est plus précise que celle des outils d'analyse classiques qui considère généralement que les interactions partiellement compatibles sont invalides, comme Wright [All97] ou Darwin [Mag99]. Dans CALICO, nous permettons un couplage très fort entre les analyses statiques et les analyses dynamiques. Ce couplage repose sur le métamodèle de débogage et sur le métamodèle d'aspect. Le métamodèle de débogage autorise les outils d'analyse statique à définir les analyses dynamiques qui doivent être effectuées pendant l'exécution du système. Le métamodèle d'aspect permet la description du mécanisme des données d'exécution requises par les analyses dynamiques.

D'autre part, nous avons choisi volontairement de ne pas fournir une solution ad-hoc pour coupler fortement les analyses statiques et dynamiques, sans perdre en pouvoir d'expression. C'est pour cela que nous avons décidé de nous baser sur le paradigme de développement orienté aspect pour définir les mécanismes d'observation des données d'exécution dans le modèle de la structure du système. De la même manière, notre métamodèle de débogage est issu des recherches existantes sur les systèmes autonomes. Il repose sur le paradigme événement/condition/action (ECA) et permet de spécifier les analyses dynamiques à réaliser pendant l'exécution du système.

Notre deuxième objectif concerne l'extensibilité du support des analyses de CALICO. Nous avons réalisé cet objectif en séparant clairement le moyen pour décrire l'architecture, c'est-à-dire les métamodèles, des outils d'analyse. Nous fournissons de plus un métamodèle d'intégration des outils d'analyse statique autorisant un expert du domaine à intégrer de nouveaux outils d'analyse et à exprimer les relations de dépendances entre toutes les analyses. Cette séparation permet à CALICO d'être extensible et générique car de nouveaux outils d'analyse peuvent être ajoutés sans que cela ait un impact sur les métamodèles et les outils existants. En outre, dans la mesure où tous nos métamodèles sont indépendants d'une plate-forme d'exécution, tous les outils d'analyse sont réutilisables pour toutes les plates-formes gérées par CALICO. CALICO permet donc de regrouper de manière uniforme, au sein d'un même canevas de développement, la plupart des outils d'analyse statique qui sont dispersés dans les divers travaux existants.

Chapitre 7

Passage de la conception à la validation dynamique

Sommaire

7.1 Introduction **125**
7.1.1 Motivation 126
7.1.2 Organisation de ce chapitre 127
7.1.3 Exemple 128
7.2 Étape d'implémentation **129**
7.2.1 Outil de génération de code 129
7.2.2 Algorithme de génération de code 130
7.3 Étape d'instrumentation **132**
7.3.1 Principe général 132
7.3.2 Événements structuraux 132
7.3.3 Événements de flot de données 133
7.3.4 Événements de QdS 135
7.4 Étape de déploiement **137**
7.4.1 Comparaison des modèles 138
7.4.2 Restructuration des opérations du modèle de mise à jour 141
7.4.3 Interprétation du modèle de mise à jour 147
7.4.4 Discussion 148
7.5 Étape de validation dynamique **149**
7.6 Conclusion **150**

Une fois que l'outil d'analyse des interactions de CALICO a vérifié la cohérence du système et qu'aucune interaction incompatible n'a été détectée, le système doit être déployé sur une plate-forme d'exécution afin de permettre sa validation dynamique par les testeurs. Ce chapitre présente les outils du support d'exécution de CALICO permettant de faire le lien entre l'étape de conception du système et l'étape de validation dynamique du système. Ces outils font la connexion entre le niveau modèle et le niveau plate-forme de CALICO.

7.1 Introduction

Cette section décrit les motivations qui nous ont guidé lors de l'élaboration des outils du support d'exécution de CALICO, puis présente l'organisation de la suite de ce chapitre.

7.1.1 Motivation

La validation dynamique d'un système nécessite l'exécution de celui-ci sur la plate-forme d'exécution cible par les testeurs. Le passage de la conception à un système qui s'exécute nécessite d'effectuer plusieurs étapes et implique la collaboration des différents acteurs élaborant le système, c'est-à-dire l'architecte logiciel, le développeur, l'administrateur système et le testeur, comme nous l'avons présenté dans la section 2.2 de l'état de l'art. D'abord les développeurs implémentent les entités du système conformément à la spécification de l'architecture. Ensuite les développeurs ont besoin d'instrumenter le système pour ajouter les assertions permettant l'exécution des analyses dynamiques. Puis, l'administrateur système déploie le système sur la plate-forme d'exécution cible. Pour finir, les testeurs exécutent les scénarios d'utilisation afin d'analyser dynamiquement les interactions partiellement compatibles. Nous souhaitons que chacune de ces étapes respecte le cahier des charges que nous avons établi en bilan de l'état de l'art dans la section 3.4.2. Nous détaillons, dans la suite, les fonctionnalités que chaque étape doit fournir.

A) Implémentation

D'une manière générale, durant l'étape d'implémentation, les développeurs écrivent le code des composants ou des services conformément aux spécifications de l'architecture. Le code d'un composant ou d'un service est composé de deux préoccupations (cf. section 2.2.4 de l'état de l'art). D'une part, le code technique est spécifique à la plate-forme à composants ou à la plate-forme orientée services et dépend fortement de la spécification du composant ou du service. Il correspond à l'implémentation des fonctionnalités requises par la plate-forme. D'autre part, le code métier est spécifique à l'application. Il correspond aux fonctionnalités du système qui sont attendues par les utilisateurs.

Une des sources d'erreurs est liée au code technique. Une simple maladresse dans l'écriture de ce code va produire un composant ou un service qui n'est pas conforme aux spécifications. De plus ce code est très redondant et extrêmement spécifique à la plate-forme d'exécution.

Notre objectif est donc de faciliter l'implémentation du système. Ainsi, nous voulons que les développeurs n'aient besoin de se focaliser que sur le code métier. Nous désirons donc que le code métier soit clairement isolé du code technique afin d'augmenter la lisibilité du code. Nous souhaitons aussi qu'il y ait un couplage fort entre la conception et l'implémentation du système afin d'éviter tout problème d'incohérence entre la spécification des entités et leur implémentation.

B) Instrumentation

D'une manière générale, pour valider dynamiquement un système, ce système doit être capable d'observer et de capturer des informations d'exécution, comme le temps de réponse ou la valeur d'une donnée qui entre dans un composant. Ces informations doivent ensuite être analysées dynamiquement afin de valider les interactions partiellement compatibles.

Malheureusement, très peu de plates-formes à composants ou orientées services existants disposent d'un support permettant l'observation d'information d'exécution. C'est donc le développeur qui doit instrumenter manuellement le code des composants et des services pour ajouter les mécanismes d'observation requis. De plus, il est aussi obligé d'ajouter les sondes de QdS nécessaires pour capturer les informations extra-fonctionnelles.

Notre objectif est donc d'automatiser complètement cette étape. Nous désirons que cette étape soit complètement transparente. De plus nous souhaitons que le mécanisme mis en place pour automatiser l'instrumentation soit extensible afin de permettre l'intégration de nouveaux canevas de sondes de QdS existants.

C) Déploiement

L'étape de déploiement consiste à instancier chaque composant et chaque service du système dans la plate-forme d'exécution, conformément aux spécifications. De manière générale, l'admi-

nistrateur système dispose de deux solutions pour déployer le système (cf. section 2.2.6). Il peut utiliser l'ADL du système qui permet le déploiement total du système, ou bien il peut utiliser l'API de déploiement fournie par la plate-forme. Cette dernière solution permet de déployer individuellement chaque composant et chaque service et autorise donc le déploiement incrémental du système.

Cependant, l'utilisation de cette API n'est pas aisée. D'une part l'API est spécifique à la plate-forme d'exécution. D'autre part, l'administrateur système a besoin d'écrire le script de déploiement manuellement en respectant les contraintes liées à l'ordre de déploiement des éléments architecturaux. De plus, le script peut devenir très volumineux, ce qui complique la tâche de l'administrateur. En outre, dans le cadre d'un déploiement incrémental, l'administrateur a besoin d'analyser manuellement les différences entre le système en cours d'exécution et le nouveau système conçu par l'architecture pour écrire son script.

Notre objectif est donc, ici aussi, d'automatiser complètement cette étape. Nous désirons que CALICO identifie automatiquement les modifications architecturales réalisées par l'architecte durant l'itération courante du cycle de développement afin de générer le script de déploiement. De plus, nous désirons que notre solution soit extensible et qu'il soit possible d'ajouter le support de nouvelles plates-formes.

D) Validation dynamique

Durant le validation dynamique d'un système, des testeurs exécutent des scénarios d'utilisation du système. Lors de l'exécution de ces scénarios, les analyses dynamiques ajoutées dans le code du système sont exécutées. Quand ces analyses détectent la violation d'une spécification, elles affichent des messages d'erreurs de très bas niveau. Généralement, elles indiquent le numéro de ligne dans le code où se trouve le problème et affiche la pile d'appel des messages.

Cependant, ces messages d'erreurs sont de trop bas niveau pour être directement compréhensibles par l'architecte. L'architecte est donc d'abord obligé d'identifier manuellement la spécification qui a été violée et qui a entraîné l'exécution de l'analyse dynamique avant de pouvoir modifier sa conception pour corriger le problème.

Notre objectif est de faciliter la tâche de l'architecte. Nous souhaitons que les messages d'erreurs générés soient directement compréhensibles par l'architecte et qu'ils indiquent l'interaction partiellement compatible qui a conduit au problème. De plus, nous désirons être génériques. Notre solution doit être indépendante de la plate-forme d'exécution utilisée.

7.1.2 Organisation de ce chapitre

Le support d'exécution de CALICO, permettant de faire le lien entre l'étape de conception et l'étape de validation dynamique, est constitué de cinq outils, comme le montre la figure 7.1.

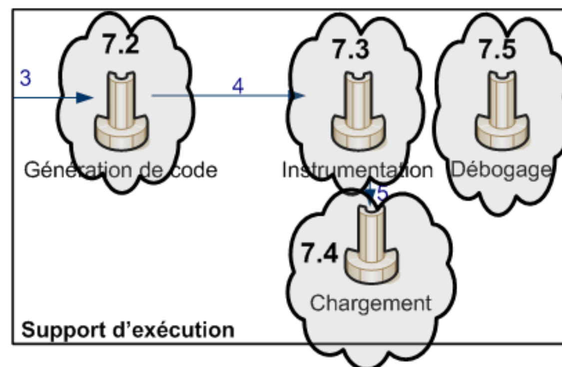

FIG. 7.1: Organisation des outils du support d'exécution de CALICO

Chaque outil est dédié à une des étapes de cycle de développement. L'étape d'implémentation est prise en charge par l'*outil de génération de code* et est détaillée dans la section 7.2. L'instrumentation du système est réalisée par l'*outil d'instrumentation*, cet outil sera présenté dans la section 7.3. L'étape de déploiement est contrôlée par l'*outil de chargement* de CALICO. Cet outil est décrit dans la section 7.4. Finalement, l'étape de validation dynamique est gérée par l'*outil de débogage* (cf. section 7.5).

7.1.3 Exemple

Durant tout ce chapitre, nous utilisons le scénario de conception du système DMP présenté dans la section 4.3.3, et plus précisément l'évolution effectuée par l'architecte lors de la seconde itération du cycle de développement, pour illustrer le passage de la conception à la validation dynamique.

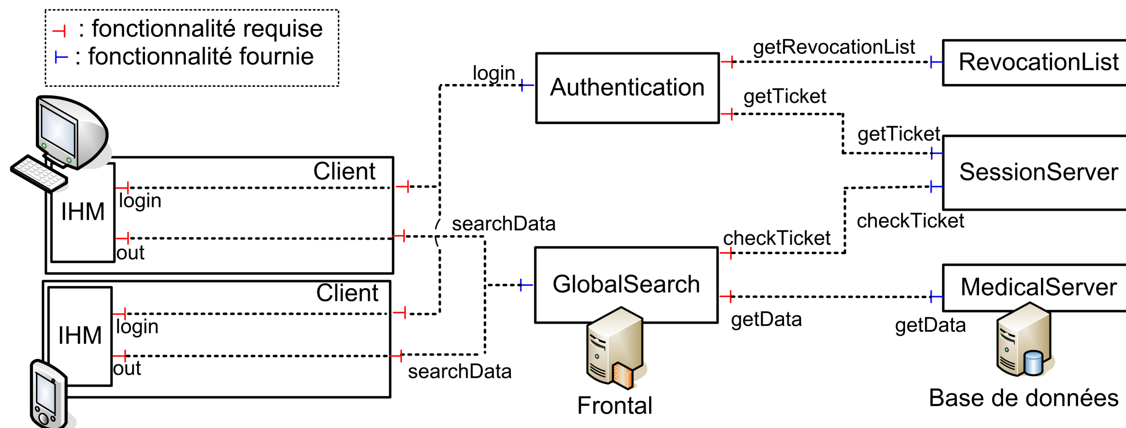

FIG. 7.2: Première version de l'architecture du système DMP

Pour rappel, dans ce scénario, l'architecte a conçu une première version de l'architecture DMP, comme cela est détaillé dans le chapitre 5 (cf. figure 7.2). Ensuite, cette conception a été analysée statiquement (cf. chapitre 6). Suite à cette analyse, l'outil d'analyse des interactions a, notamment, identifié que l'interaction entre le PDA et la base de données est partiellement compatible. En effet, d'une part, le PDA ne peut pas recevoir de données médicales plus grandes que 10 mégaoctets. D'autre part, il ne peut recevoir que des images de type JPG ou du texte ASCII. En conséquence, l'analyse statique a ajouté des analyses dynamiques dans le modèle de débogage.

Il est donc maintenant nécessaire que les testeurs effectuent une validation dynamique du système DMP. Pour permettre cette validation dynamique, les outils du support d'exécution déploient le système et mettent en œuvre les mécanismes requis pour observer les données d'exécution. Ainsi, les testeurs peuvent, par exemple, jouer les scénarios d'utilisation d'un pharmacien et d'un radiologue. Comme cela est expliqué dans la section 4.3.3, les testeurs détectent qu'un radiologue ne peut pas visionner de radiographies sur le PDA car celles-ci sont trop volumineuses. Afin de résoudre ce problème, l'architecte ajoute une entité `DataConverter` entre le PDA et la base de données dont le rôle est de réduire la taille des images (cf. figure 7.3).

La seconde version de l'architecture DMP est ensuite analysée statiquement par l'outil d'analyse des interactions des CALICO. Grâce aux garanties fournies par l'entité `DataConverter`, l'analyse statique garantit que cette seconde version de l'architecture ne viole pas la propriété liée à la taille de la donnée médicale. Néanmoins, l'interaction est toujours partiellement compatible car cette seconde version de l'architecture n'apporte aucune garantie sur le fait que le PDA recevra des données de type JPG. En conséquence l'analyse statique ajoute une analyse dynamique dans le modèle de débogage afin de permettre la validation dynamique de l'interaction.

Afin de valider dynamiquement le logiciel, les outils du support d'exécution de CALICO

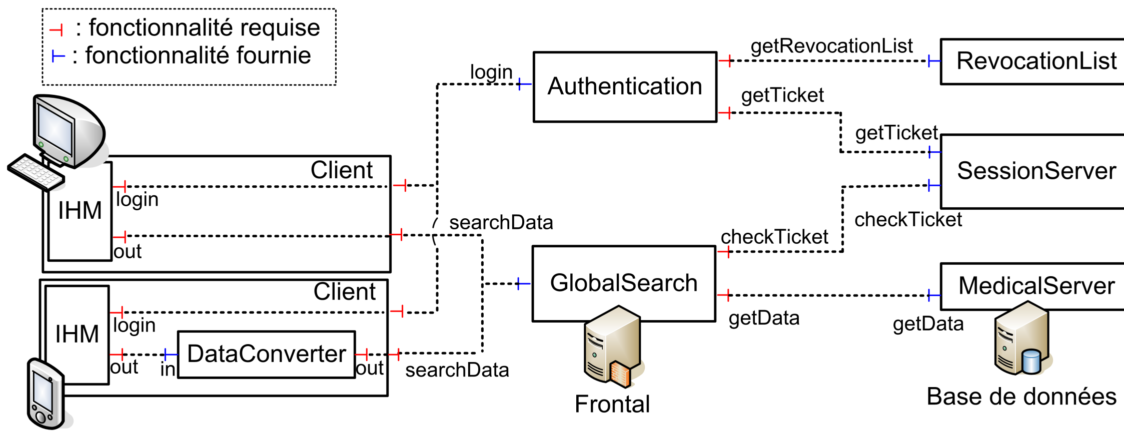

FIG. 7.3: Seconde version de l'architecture du système DMP

propagent l'évolution décrite par l'architecte dans le logiciel en cours d'exécution. Nous utilisons cette propagation vers la plate-forme d'exécution Fractal pour illustrer nos propos dans ce chapitre.

7.2 Étape d'implémentation

Après la conception et la validation de l'architecture, l'étape de développement suivante est l'étape d'implémentation. Durant cette étape, les développeurs écrivent le code métier et technique des composants et des services, en accord avec la spécification de l'architecture.

L'objectif de l'étape d'implémentation de CALICO est de générer complètement le code technique des composants et des services qui n'existent pas, en fonction de la plate-forme d'exécution cible. Cette tâche est réalisée automatiquement par l'outil de génération de code de CALICO.

Dans le cas d'une conception qui repose exclusivement sur une intégration de composants et de services existants, l'étape d'implémentation n'est pas effectuée. Le cycle de développement est alors un cycle uniforme et ininterrompu entre l'étape de conception et l'étape de validation dynamique, ce qui permet d'augmenter la vitesse de mise au point du logiciel.

7.2.1 Outil de génération de code

L'outil de génération de code est extensible et gère différentes plates-formes. Ce support repose sur le métamodèle d'intégration de CALICO représenté dans la figure 7.4. Ce métamodèle

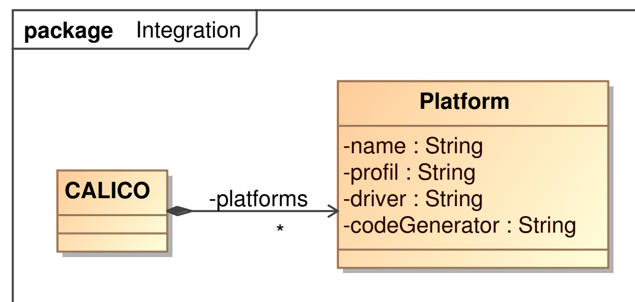

FIG. 7.4: Métamodèle d'intégration

permet d'associer dans chaque plate-forme (`Platform`) le nom de la classe d'exécution du driver de génération de code (`codeGenerator`). Le driver repose, quant à lui, sur les outils classiques de transformation de modèles vers du texte, comme par exemple Xpand [1].

Cette solution permet d'étendre CALICO. Ce dernier permet à un expert de la plate-forme d'ajouter le support de génération de code d'une nouvelle plate-forme en écrivant un driver et en mettant à jour le modèle d'intégration. Par exemple, dans le cas d'une utilisation de Xpand, le driver correspond à un fichier *template*.

7.2.2 Algorithme de génération de code

Une fois le driver sélectionné en fonction de la plate-forme cible, CALICO exécute ce driver. Le driver a pour rôle de générer le squelette de code, le fichier ADL et l'implémentation des composants ou des services si le driver dispose de suffisamment d'informations.

A) Génération du squelette de code

Pour générer le squelette de code, le driver utilise les informations stockées dans le modèle de la structure du système. Pour chaque entité primitive, il récupère, à partir des attributs stockés dans le modèle de l'entité, le nom de la classe d'implémentation. Ensuite, à partir des informations sur les points de communication requis et fournis, le driver génère le code technique, comme par exemple les annotations du modèle Fractal ou les annotations du modèles des Services WEB. Les développeurs n'ont alors besoin que de coder le code métier des composants et des services et de le compiler en utilisant leurs outils de développement habituel, comme Eclipse.

Globalement, ce mécanisme permet de générer tout le code technique des composants et des services WEB en fonction des informations spécifiées par l'architecte au niveau modèle. De plus, le code technique est regénéré chaque fois que l'architecte modifie sa conception. Ce mécanisme permet donc de garantir que l'implémentation est toujours conforme à la spécification. En outre, dans la mesure où les informations qui sont requises par l'outil d'implémentation sont spécifiées dans le profil de plate-forme, alors l'outil d'implémentation a la garantie que ces informations existent dans le modèle de la structure du système.

Exemple. Dans notre scénario, CALICO génère le squelette de code de la nouvelle entité `DataConverter`, représentée dans la figure 7.3, pour la plate-forme Fractal choisie par l'architecte (cf. listing 7.1). Les lignes 1 à 4 correspondent au code du point de communication `in` de type `SearchDataItf`. Dans la plate-forme Fractal Julia, le code d'un point de communication est une interface Java annotée avec `@Interface`. La ligne 3 code l'élément de communication `searchPicture` avec son argument en entrée `session` et son argument en

[1] http ://www.eclipse.org/modeling/m2t/

```
1 @Interface(name="in")
2 public interface SearchDataItf {
3   public MedicalInformation searchPicture(Session session);
4 }

6 @Component(name="DataConverter" , provides=@Interface(name="in", signature=SearchDataItf))
7 public class DataConverterImpl implements SearchDataItf {
8   @Requires(name="out")
9   private SearchDataItf out;

11   public MedicalInformation searchPicture(Session session) {
12     ...
13     return null;
14   }
15 }
```

LST. 7.1: Squelette de code de l'entité `DataConverter`

sortie `MedicalInformation`. Les lignes 6 à 15 correspondent à l'implémentation de l'entité `DataConverter`. Cette implémentation est une classe Java annotée avec `@Component`. Le nom de la classe d'implémentation est récupéré dans l'attribut `impl-class` de l'entité (cf. le profil de la plate-forme Fractal dans la section 5.2.2). Les points de communication requis correspondent à des attributs Java annotés avec `Requires` (cf. lignes 8 et 9). Ainsi, les développeurs peuvent compléter ce squelette de code afin d'implémenter le métier du composant (cf. ligne 12).

B) Génération de l'implémentation

Au niveau modèle, l'architecte peut décrire le flot de contrôle des entités. Ce flot de contrôle décrit l'ordre d'enchaînement de réceptions et d'envois des messages, ce qui correspond à une orchestration dans les architectures orientées services. Or, il est possible à partir de ces informations comportementales de générer automatiquement le code des entités. Par exemple, ce mécanisme est très utilisé dans le monde des services WEB pour générer le fichier BPEL en fonction du diagramme BPMN ou pour produire le code Java à partir d'un fichier BPEL. CALICO permet donc de réutiliser ces outils de génération de code afin de générer le code des orchestrations.

C) Génération de l'ADL

Généralement, les modèles à composants proposent un ADL permettant de décrire l'assemblage des composants. Cette description de l'assemblage est ensuite typiquement utilisée pour déployer tout le système. Le driver de génération de code génère aussi les fichiers ADLs en fonction des informations contenues dans les modèles. Cette fonctionnalité de CALICO permet, une fois le développement du système fini, de produire le fichier ADL afin de permettre la mise en production du système en dehors du canevas CALICO.

Exemple. Le listing 7.2 est un extrait de l'ADL du système DMP généré pour la plate-forme Fractal. Les lignes 1 à 3 définissent l'entête du fichier ADL. Les lignes 5 à 19 décrivent le composant composite Fractal du système DMP. Les lignes 8 à 17 correspondent à l'entité composite `Client` du PDA. Les lignes 10 à 14 représentent l'entité `DataConverter`. Les lignes 11 et 12 définissent ces points de communication requis et fournis et la ligne 13 spécifie sa classe d'implémentation `DataConverterImpl` qui a été généré précédemment.

```
1<?xml version="1.0" encoding="ISO-8859-1" ?>
2<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
3    "classpath://org/objectweb/fractal/adl/xml/basic.dtd">

5<definition name="DMP">
6  <interface name="r" role="server" signature="java.lang.Runnable"/>
7   ...
8  <component name="Client">
9     ...
10    <component name="DataConverter">
11      <interface name="in"      role="server"  signature="DMP.SearchDataItf"/>
12      <interface name="out"     role="client"  signature="DMP.SearchDataItf"/>
13      <content class="DataConverterImpl"/>
14    </component>
15    <binding client="IHM.out" server="DataConverter.in"/>
16     ...
17  </component>
18    ..
19</definition>
```

LST. 7.2: Extrait de l'ADL du système DMP

7.3 Étape d'instrumentation

Afin de valider dynamiquement un système, il est nécessaire d'observer et de capturer les données d'exécution. L'objectif de l'étape d'instrumentation est de permettre cette observation, indépendamment de la plate-forme d'exécution.

Cette tâche est réalisée automatiquement et de manière transparente par l'outil d'instrumentation de CALICO. Cet outil instrumente le code des composants et des services implémentés par les développeurs afin de permettre au logiciel de faire remonter les données d'exécution au niveau modèle.

7.3.1 Principe général

L'outil d'instrumentation de CALICO a pour rôle de permettre la capture des données d'exécution requises par les analyses dynamiques décrites dans le modèle de débogage. De plus, il prend en charge le développement incrémental du système. Cela signifie qu'entre chaque itération, de nouvelles analyses dynamiques peuvent apparaître et d'autres peuvent être supprimées. Donc, l'outil ajoute ou supprime des sondes dans le système en fonction des analyses dynamiques contenues dans le modèle de débogage.

Afin d'instrumenter le système, l'outil utilise la démarche suivante : en entrée, l'outil dispose de deux versions du modèle de débogage. La première version correspond à l'ancien modèle de débogage issu de l'itération précédente du cycle de développement. La seconde version est le modèle de débogage provenant de l'itération courante du cycle de développement. L'outil d'instrumentation effectue une comparaison des deux modèles afin d'identifier les analyses dynamiques qui ont été ajoutées et celles qui ont été enlevées. Pour chacune des analyses dynamiques, cet outil en déduit le type de l'événement associé à la donnée d'exécution qui doit être observée. En fonction du type de cet événement, l'outil exécute un plugin de telle sorte que : si l'analyse dynamique est supprimée, le plugin supprime aussi la sonde associée du système ; si l'analyse dynamique est nouvelle, le plugin instrumente le système pour permettre la capture des données requises.

Dans la suite de cette section, nous détaillons le mécanisme d'instrumentation pour chacun des types d'événements du modèle de débogage, c'est-à-dire structuraux, de flot de données et de QdS.

7.3.2 Événements structuraux

Il existe quatre types d'événements structuraux décrivant les modifications de la structure du système. L'observation de ces événements consiste à détecter les ajouts et les suppressions d'entités et de connecteurs dans la plate-forme. Cependant très peu de plates-formes fournissent nativement un mécanisme d'observation de ces modifications.

Afin de pallier la lacune de ces plates-formes, nous proposons une approche générique pour capturer les modifications structurelles. Nous partons du constat que, dans les plates-formes à composants et dans les plates-formes orientées services, ces modifications structurelles correspondent toujours au résultat d'une invocation de l'API de déploiement de la plate-forme. En conséquence, en interceptant ces appels à l'API de déploiement, il est possible de connaître les modifications structurelles avant que celles-ci ne soient appliquées aux systèmes. Pour réaliser ces interceptions, notre approche se base sur le développement orienté aspect [KLM+97].

Notre approche consiste donc à générer et à tisser quatre aspects dans le code du système. Chacun des aspects a pour rôle de capturer un type d'événement structurel, ils sont tissés juste avant l'appel à l'API de déploiement qui correspond à la modification structurelle à observer. Les événements structurels capturés sont tous remontés au niveau métier. Par exemple, l'aspect correspondant à l'observation d'ajout d'entité est tissé avant chaque appel à l'API d'ajout d'une entité.

Globalement cette approche est utilisable sur toutes les plates-formes à composants ou à services dans la mesure où il existe un tisseur d'aspect pour le langage de programmation de la plate-forme. De plus, cette approche ne modifie pas la plate-forme car les aspects sont tissés dans le code métier de l'application afin d'intercepter les appels à l'API de déploiement. Mais aussi, cette solution permet de ne capturer que les modifications structurelles effectuées par le système lui même. De cette manière, les modifications structurelles provenant des propagations des changements de conception faites par l'architecte, au niveau modèle, dans la plate-forme ne sont pas observées.

7.3.3 Événements de flot de données

Il existe deux types d'événement de flot de données : `MessageEnter` et `MessageExit` qui correspondent respectivement à un événement de réception et d'envoi des messages m_i par un point de communication P (cf. section 6.2.1). L'observation de ces deux événements consiste à faire remonter au niveau modèle la valeur des messages m_i. Cependant, cette méthode d'observation ne prend pas en compte le flot de contrôle, c'est-à-dire la trace des messages échangés, qui peut être requise pour certaines analyses statiques, comme par exemple l'analyse de flot de données. En conséquence, la méta-classe `DataflowEvent` possède un attribut précisant s'il faut aussi capturer la trace.

A) Observation sans prise en compte de la trace

Afin de permettre l'observation d'un événement de flot de données e, l'outil d'instrumentation ajoute un aspect A sur le point de communication P. L'aspect est tissé avant ou après l'appel des éléments de communication de P tel que :

- $A.type = BEFORE$ si P est fourni et que e est un événement de réception, ou si P est requis et que e est un événement d'envoi.
- $A.type = AFTER$ si P est fourni et que e est un événement d'envoi, ou si P est requis et que e est un événement réception.

Cet aspect intercepte les appels et les retransmet à l'entité advice `ReifyValue` qui code la préoccupation transverse. Cette entité advice fait remonter au niveau modèle les messages m_i et bloque l'appel en cours en attendant le résultat des analyses dynamiques faites au niveau modèle. Si aucune erreur n'est détectée, l'entité continue l'appel. Si une erreur survient, l'entité arrête la transmission de l'appel et renvoie une valeur de retour par défaut qui a été spécifiée par l'architecte au niveau modèle. Le but étant de retourner une valeur neutre pour ne pas faire effondrer le système en cours d'exécution. Le code de cette entité `ReifyValue` est généré par l'outil d'instrumentation.

Dans les cas où l'analyse dynamique qui nécessitait l'observation de l'événement de flot de données a été supprimée, CALICO supprime l'aspect capturant cet événement. De cette manière, seuls les événements de flot de données requis sont observés.

Exemple. Dans notre scénario, le PDA ne peut pas recevoir des images médicales qui ne sont pas de type JPG. L'analyse du flot de données dans le système DMP a donc détecté que l'interaction avec le PDA est partiellement compatible. L'analyse statique a alors ajouté une analyse dynamique (`Rule r1`) dans le modèle de débogage, qui est représentée dans la figure 7.5. Dans cet exemple, l'analyse dynamique a besoin de connaître les données `data` qui sont envoyées par le point de communication `getData` de l'entité `MedicalServer`.

L'outil d'instrumentation ajoute alors un aspect `aspectD1` sur le point de communication `getData` de l'entité `MedicalServer` pour intercepter les données `data` sortantes, comme le montre le modèle de la figure 7.5. Le rôle de l'entité d'aspect `ReifyValue` est de propager la valeur de la donnée au niveau modèle afin d'effectuer l'analyse dynamique du type de la donnée.

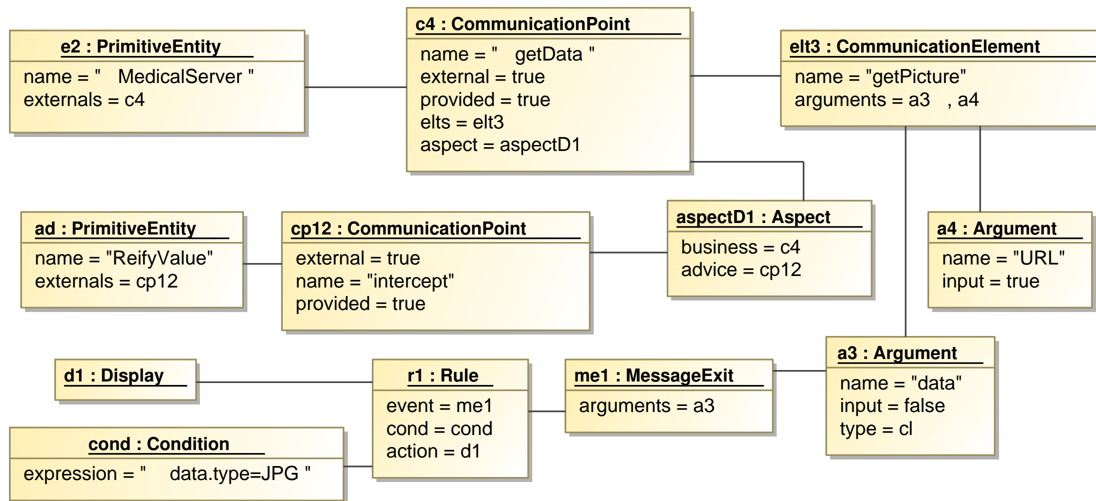

FIG. 7.5: Aspect associé à l'analyse dynamique du flot de données

B) Observation avec prise en compte de la trace

L'observation d'un événement de flot de données avec la trace est similaire à l'observation de l'événement sans la trace, sauf qu'en plus, l'observation doit prendre en considération la trace des messages. Cependant, la plupart des plates-formes à composants et orientées services ne fournissent pas de support natif pour capturer la trace.

Donc, afin de capturer cette trace, la liste des messages entrants et sortants de chaque entité du système a besoin d'être collectée. Concrètement, il est nécessaire de prendre en compte le flot de contrôle interne à chaque entité du système. Donc, l'implémentation des entités, c'est-à-dire le code source, doit être modifiée. Une fois de plus, CALICO utilise une approche orientée aspect.

Notre approche repose sur la génération et le tissage de trois aspects. Ces aspects sont générés à partir des informations contenues dans le modèle de débogage et le modèle de structure du système. Pour expliquer le fonctionnement de nos aspects, nous utilisons l'exemple d'instrumentation de l'entité `DataConverter` du DMP qui a été ajoutée par l'architecte, comme le montre le listing 7.3. L'instrumentation se focalise sur l'élément de communication `searchPicture` du point de communication `in`. Le premier aspect permet à une entité de recevoir la trace.

```
1 @Component(name="DataConverter" , provides=@Interface(name="in", signature=SearchDataItf))
2 public class DataConverterImpl implements SearchDataItf {
3   @Require(name="out")
4   private GetDataItf out;

6   public WrappedResult searchPicture(Session session, List<String> trace) {
7     trace.add("?DataConverter.in.searchPicture.session");
8     ...
9     trace.add("!DataConverter.out.getPicture.URL");
10    WrappedResult __res = this.out.getPicture(URL, trace);
11    trace = __res.getTrace();
12    trace.add("?DataConverter.out.getPicture.data");
13    ...
14    trace.add("!DataConverter.in.searchPicture.data")
15    return new WrappedResult(resultImage, trace);
16  }
17 }
```

LST. 7.3: Instrumentalisation de l'entité `DataConverter`

Concrètement, il ajoute un paramètre `trace` de type liste à tous les éléments de communication se trouvant dans les points de communication fournis (cf. ligne 6 du listing 7.3). Il sauvegarde aussi dans la liste `trace` l'activité correspondant à la réception d'un message, sous forme d'expression SFSP (cf. ligne 7).

Le second aspect permet à une entité de renvoyer la trace. Concrètement, il remplace le type de retour, de chaque élément de communication se trouvant dans les points de communication fournis par un nouvel objet de type `WrappedResult` qui encapsule à la fois le résultat de l'élément de communication et la trace (cf. ligne 6). Il ajoute dans la liste l'activité correspondant à l'envoi du message de retour (cf. ligne 14) et retourne l'objet qui encapsule le résultat (cf. ligne 15).

Le troisième aspect permet à une entité d'envoyer et de recevoir la trace quand elle appelle des points de communication fournis des autres entités. Pour réaliser cette retransmission de la trace, juste avant chaque instruction d'invocation, l'aspect ajoute dans la liste l'activité correspondant à l'envoi de messages (cf. ligne 9). Après chaque instruction d'invocation, il ajoute dans la liste l'activité correspondant à la réception de messages (cf. ligne 12). Cet aspect modifie aussi l'instruction d'invocation afin de passer en paramètre la trace et de récupérer en retour la trace calculée par l'entité invoquée (cf. ligne 10). Ensuite il utilise cette nouvelle trace (cf. ligne 11).

Une fois les aspects tissés dans le système, l'application est capable de calculer la trace des messages. Cette trace est envoyée au niveau modèle par l'entité `ReifyValue` exactement de la même manière qu'un message métier.

7.3.4 Événements de QdS

D'une manière générale, l'observation des événements de QdS nécessite que le développeur intègre dans son système des sondes de QdS, en fonction du type de l'événement de QdS requis par les analyses dynamiques. L'outil d'instrumentation de CALICO effectue cette tâche automatiquement. Pour chaque événement de QdS, il identifie le type T de l'événement grâce à l'association `type` entre un événement de QdS (`QoSEvent`) et un type de QdS (`QoSType`) du métamodèle de débogage (cf. section 6.2.1). Ensuite, à partir de ce type T, l'outil détermine le type de sonde de QdS à ajouter dans le système.

Ce support repose sur le métamodèle d'intégration de la figure 7.6. Ce métamodèle permet de faire le lien entre un type de QdS (`QoSType`), décrit au niveau modèle, et un plugin d'instrumentation de QdS (`Instrumentation`) capable d'ajouter la sonde de QdS qui est requise pour observer l'événement de QdS. De plus, pour être indépendant de toute plate-forme, chaque plugin d'instrumentation est associé à une méta-classe `Platform`, qui représente une plate-forme d'exécution à composants ou orientée services. Cette approche permet à CALICO d'instancier la sonde QdS correcte en fonction du type de QdS et de la plate-forme sous-jacente.

En outre, le métamodèle d'intégration permet à CALICO d'être extensible. En effet, un expert du domaine de la QdS peut ajouter la gestion de nouvelles sondes de QdS en modifiant le modèle d'intégration. Il peut ainsi décrire quel plugin d'instrumentation doit être exécuté en fonction du type de QdS et de la plate-forme.

Un plugin d'instrumentation possède une classe d'implémentation qui correspond à la classe du plugin à exécuter pour effectuer l'instrumentation. La classe du plugin est composée de deux méthodes : une méthode pour ajouter une sonde de QdS et une méthode pour enlever la sonde du système. Ces méthodes prennent en entrée un objet de type `QoSEvent` correspondant à l'événement de QdS qui doit ou devait être observé. Le plugin a accès à tous les modèles de CALICO, ce qui lui permet de calculer la localisation où la sonde doit être ajoutée, en fonction de l'événement à capturer. Les actions effectuées par un plugin d'instrumentation peuvent être variées en fonction du type de QdS à observer. Par exemple, le plugin peut ajouter un aspect dans le modèle d'aspect et générer le code de l'entité correspondant à la fonctionnalité transverse. Il peut aussi modifier le code des composants et des services à l'aide d'un tisseur d'aspect.

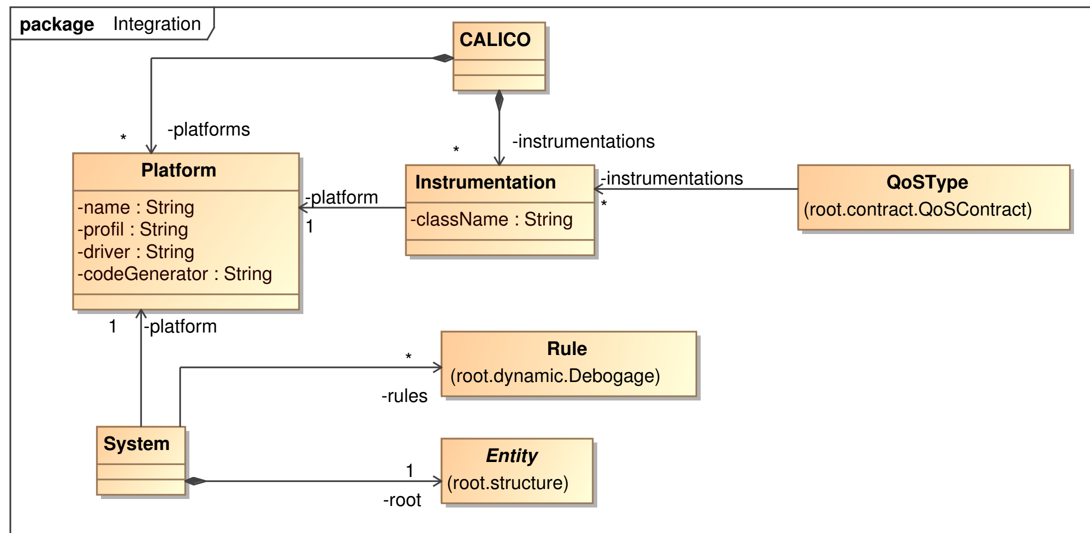

FIG. 7.6: Métamodèle d'intégration

```
1 context System
2   inv: self.rules->forAll(r:Rule | r.event.oclIsTypeOf(QoSEvent) implies
3         r.event.oclAsType(QoSEvent).type.instrumentations->exists(
4         i:Instrumentation | i.platform = self.platform)
```

LST. 7.4: Contraintes OCL du métamodèle d'intégration

Le métamodèle d'intégration sert aussi à spécifier le système (`System`). Un système contient l'architecture conçue par l'architecte (association `root`), l'ensemble des analyses dynamiques et une référence vers la plate-forme d'exécution cible. Afin de garantir que les événements de QdS peuvent être observés dans la plate-forme cible, des contraintes OCL sont ajoutées sur le métamodèle d'intégration, comme le montre le listing 7.4. Ces contraintes vérifient, que pour chaque type de données de QdS requises par les analyses dynamiques, il existe toujours un plugin d'instrumentation utilisable sur la plate-forme cible. Si un plugin d'instrumentation est manquant, CALICO le signale à l'architecte dès l'étape de conception. De cette manière l'architecte est averti, le plus tôt possible, que l'analyse dynamique correspondante ne sera pas prise en compte durant la validation dynamique du système logiciel.

Globalement, le modèle d'intégration est le modèle clé de CALICO. Il garde le lien entre la spécification de l'architecture faite au niveau modèle et les sondes requises dans la plate-forme. L'extensibilité de CALICO repose aussi sur ce modèle qui permet l'ajout de nouvelles sondes. De plus, il permet à l'outil d'instrumentation de CALICO d'insérer ou enlever les sondes de QdS afin de capturer uniquement les événements de QdS requis pour finaliser l'analyse des interactions. Ce mécanisme évite ainsi le surcoût lié à l'observation d'événements de QdS inutiles.

Exemple. Dans notre scénario, les périphériques clients doivent afficher les informations médicales en moins de 10s. Dans la mesure où cette propriété applicative ne peut pas être vérifier statiquement, l'outil d'analyse des interactions de CALICO a ajouté une analyse dynamique (concept `Rule`) dans le modèle de débogage (cf. section 5.3.5). Cette analyse a besoin de connaître la valeur de l'événement de QdS de type `''MaximalResponseTime''` (concept `QoSType`). Dans cet exemple, l'application DMP est implémentée avec le modèle à composants Fractal. En conséquence, l'outil d'instrumentation récupère, à partir du modèle d'intégration, le plugin d'instrumentation `Instrumentation` qui est dédié à l'insertion, dans une plate-forme Fractal, d'une

sonde capturant le temps de réponse. Une fois que le plugin est trouvé, l'outil d'instrumentation invoque ce plugin afin d'intégrer la sonde de QdS dans le système.

7.4 Étape de déploiement

Une fois que le système est instrumenté, l'étape suivante consiste à déployer ce système sur la plate-forme cible afin de l'exécuter. Généralement, pour réaliser cette étape, l'administrateur système a besoin d'écrire un script de déploiment décrivant la succession des opérations de déploiement à exécuter, comme par exemple déployer ou replier [2] une entité.

Cette tâche est automatiquement réalisée par l'outil de chargement de CALICO (cf. figure 7.7). Nous considérons que la plate-forme a déjà été déployée par l'administrateur système. L'outil de chargement se focalise donc sur le déploiement de l'application.

CALICO est un canevas de développement incrémental et itératif. En conséquence, à chaque itération, des éléments architecturaux peuvent être ajoutés ou supprimés par l'architecte. Il est

[2] C'est l'action inverse du déploiement qui consiste à retirer une entité de la plate-forme.

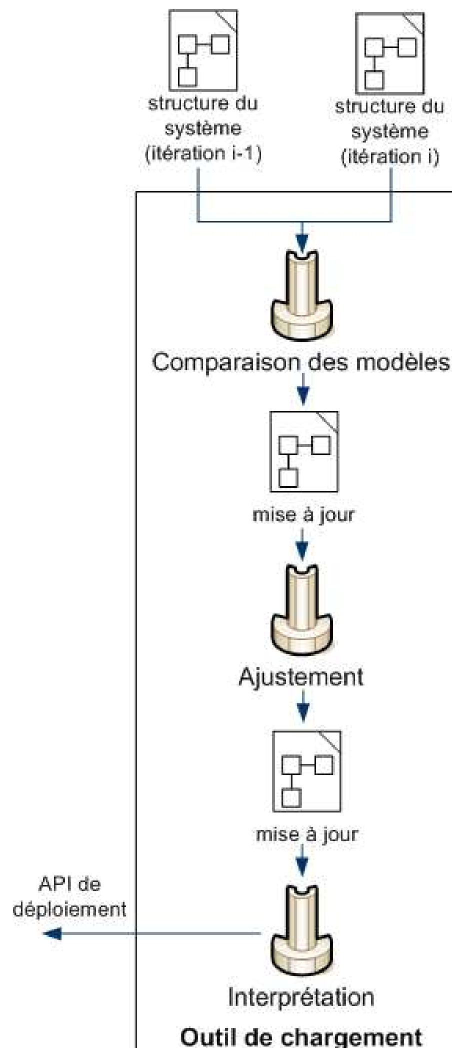

FIG. 7.7: Organisation de l'outil de chargement

donc nécessaire de propager ces changements dans le système en cours d'exécution. Donc l'outil de chargement a besoin de prendre en compte le déploiement incrémental d'un système. De plus, nous souhaitons que cet outil soit extensible afin de pouvoir gérer de multiples plates-formes d'exécution différentes.

Afin d'atteindre cet objectif, nous avons décomposé le déploiement d'une application en trois étapes, comme le montre la figure 7.7. La première étape est l'étape de comparaison des modèles de structure du système. Elle consiste à identifier les changements architecturaux effectués par l'architecte dans le cycle de développement courant afin de produire un script de déploiement, qui est appelé modèle de mise à jour, indépendant de toute plate-forme. La seconde étape est l'étape d'ajustement. Durant cette étape, le modèle de mise à jour est ajusté en fonction, du support de déploiement offert par la plate-forme cible, et en fonction des contraintes liées à l'ordonnancement des opérations de déploiement. La troisième étape est l'étape d'interprétation du modèle de déploiement. Durant cette étape, l'outil de chargement appelle l'API de déploiement de la plate-forme cible.

7.4.1 Comparaison des modèles

L'objectif de cette étape est d'identifier les changements architecturaux qui ont été effectués entre deux itérations du cycle de développement. Cette tâche est effectuée par le moteur de comparaison des modèles (cf. figure 7.7). Ce moteur effectue une transformation de modèle. Il prend en entrée deux versions du modèle de la structure du système. La première version du modèle correspond à la structure du système de l'itération précédente, c'est-à-dire qu'il décrit l'architecture du système qui est en cours d'exécution. La seconde version correspond à la nouvelle conception effectuée par l'architecte dans l'itération courante du cycle de développement. Le moteur de comparaison analyse la différence entre ces deux modèles afin de produire en sortie un modèle de mise à jour, conforme au métamodèle de mise à jour de la figure 7.8.

A) Métamodèle de mise à jour

Le métamodèle de mise à jour permet de décrire les règles de transformation du système, ainsi que leur ordre d'application afin de propager les modifications effectuées au niveau modèle dans le système en cours d'exécution. Ce métamodèle exprime la séquence des opérations de construction du système, comme par exemple, ajouter une entité ou supprimer un connecteur (cf. figure 7.8).

Un script de mise à jour (`UpdateRules`) est composé d'un ensemble ordonné d'opérations de construction (`Update`). Ces opérations de construction sont très élémentaires afin de décrire le plus précisément possible les actions de modification à effectuer. Pour décrire la sémantique de chacune de ces actions, nous appelons $System$ la version du modèle de structure du système de l'itération précédente, c'est-à-dire celui qui décrit la structure du système qui est en cours d'exécution. Nous nommons $Design$ la version courante du modèle de structure du système, c'est-à-dire celui qui correspond à la nouvelle conception de l'architecture. Par exemple, dans notre scénario DMP, $System$ correspond à l'architecture décrite dans la figure 7.2 et $Design$ correspond à l'architecture représentée dans la figure 7.3.

Les opérations d'introduction sont utilisées pour modifier localement la structure d'une entité. L'opération `AddCommunicationPoint` permet d'ajouter un nouveau point de communication `elt`, défini dans le modèle $Design$, sur l'entité référencée par `on` dans le modèle $System$. Ce nouveau point de communication peut posséder des éléments de communication.

L'opération `RemoveCommunicationPoint` supprime le point de communication référencé par `elt`, défini dans le modèle $System$. L'entité qui possède ce point de communication est retrouvée en naviguant le modèle $System$. Ce point de communication peut posséder des éléments de communication.

L'opération `AddParameter` ajoute un nouveau paramètre `elt`, défini dans le modèle $Design$, sur l'entité référencé par `on` dans le modèle $System$.

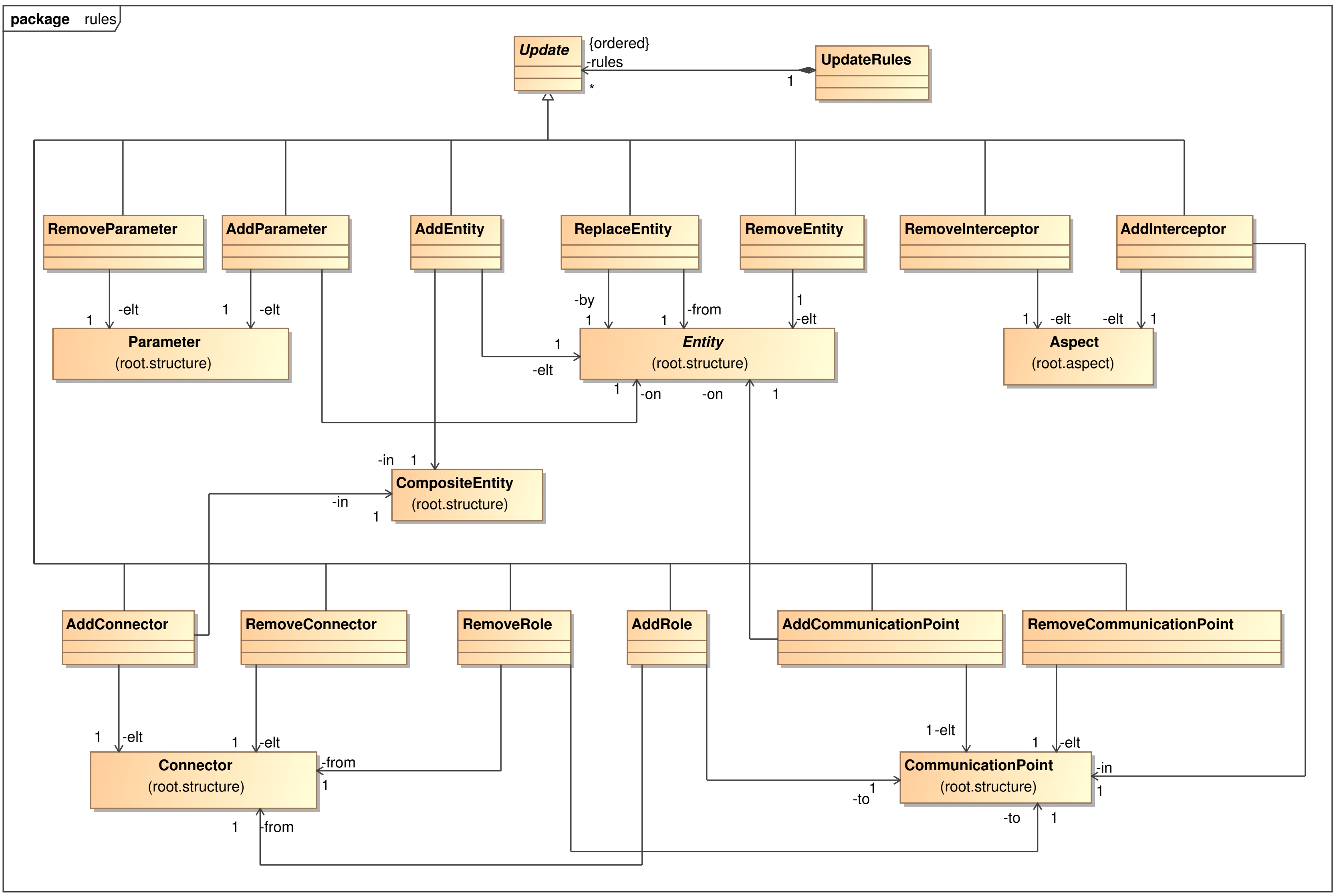

FIG. 7.8: Métamodèle de mise à jour

L'opération `RemoveParameter` supprime le paramètre `elt`, défini dans le modèle $System$. L'entité peut être retrouvée en parcourant le modèle $System$.

Les opérations de reconfiguration concernent la modification des interactions entre les entités. L'opération `AddEntity` ajoute la nouvelle entité référencée par `elt`, décrite dans le modèle $Design$, dans le composite référencé par `in` spécifié dans le modèle $System$. Cette entité peut posséder des points de communication. Elle peut aussi être primitive ou composite. Dans le cas d'une entité composite, cette opération n'ajoute pas les entités contenues dans le composite.

L'opération `RemoveEntity` supprime l'entité référencée par `elt` définie dans le modèle $System$. Cette entité peut posséder des points de communication. Si cette entité est une entité composite, elle doit absolument ne contenir aucune entité.

L'opération `ReplaceEntity` remplace l'entité du modèle $System$, référencée par `from`, par l'entité du modèle $Design$ référencée par `by`. Cette entité doit obligatoirement être une entité primitive.

L'opération `AddConnector` supprime le connecteur du modèle $system$ référencé par `elt`. L'entité composite possédant ce connecteur peut être retrouvée en naviguant dans le modèle $System$.

L'opération `AddRole` permet de connecter le point de communication du modèle $System$ référencé par `to` avec le connecteur du modèle $System$ référencé par `from`.

L'opération `RemoveRole` permet de déconnecter le point de communication du modèle $System$ référencé par `to` avec le connecteur du modèle $System$ référencé par `from`.

Les opérations de tissage permettent de tisser les aspects sur les points de communication. L'opération `AddInterceptor` tisse l'aspect du modèle $Design$ référencé par `elt` sur le point de communication du modèle $System$ référencé par `in`.

L'opération `RemoveInterceptor` supprime l'aspect du modèle $System$ référencé par `elt`. Le point de communication impliqué par l'aspect peut être retrouvé en naviguant dans le modèle $System$.

B) Algorithme

Dans notre canevas, nous avons besoins de comparer deux modèles qui sont conforment au même métamodèle. Il existe de nombreuses approches génériques pour comparer deux modèles [KPP06]. Néanmoins, dans notre approche nous ne souhaitons pas calculer toutes les différences entre les deux modèles. Nous souhaitons que le résultat de la comparaison ait une sémantique propre au domaine des architectures logicielles, c'est-à-dire que la comparaison produise le modèle de mise à jour qui contient les concepts propres à la reconfiguration d'architectures. Nous avons donc élaboré un algorithme qui est spécifique au domaine en spécialisant un algorithme généraliste.

Exemple. Dans notre scénario, pour propager les évolutions effectuées par l'architecture lors de l'élaboration de la seconde version du DMP (cf. figure 7.3), le moteur de comparaison des modèles compare l'architecture de la figure 7.2 et celle de la figure 7.3. Il obtient le modèle de mise à jour qui contient la liste des opérations de construction du système suivante :

```
AddConnector between DataConverter.out and PDAClient.searchData
AddConnector between IHM.out and DataConverter.in
AddEntity DataConverter in PDAClient
RemoveConnector between IHM.out and PDAClient.searchData
```

Pour des raisons de lisibilité, nous avons représenté dans cet exemple la liste des opérations de construction du système de manière textuelle. Cette liste exprime que pour propager l'évolution dans le système en cours d'exécution, il est nécessaire d'ajouter deux connecteurs, d'ajouter la nouvelle entité `DataConverter` dans l'entité `PDAClient` et de supprimer l'ancien connecteur.

C) Discussion

CALICO gère de manière uniforme et sans distinction les évolutions qui correspondent à des ajouts et des suppressions de fonctionnalités. En effet, le mécanisme de déploiement repose sur un modèle de mise qui est issu de la comparaison de deux versions du modèle de structure du système. Ainsi, ce modèle exprime l'ensemble des opérations de construction à effectuer pour passer de la première version du modèle à la seconde version, indépendamment de la nature de l'évolution effectuée entre ces deux versions. De plus, puisque les deux modèles sont cohérents, car ils ont été tous les deux validés par l'outil d'analyse des interactions, le modèle de mise à jour obtenu engendrera obligatoirement une seconde version cohérente du modèle de la structure du système.

Notre métamodèle de mise à jour est à granularité extrêmement fine. Il est capable de capturer des modifications structurelles, telle qu'un ajout ou une suppression de point de communication, indépendamment de toute plate-forme. De cette manière, notre métamodèle n'est pas soumis aux limitations des plates-formes. Cette solution augmente l'extensibilité de CALICO et permet à CALICO de s'adapter à la finesse des APIs de déploiement des plates-formes. Par exemple, si dans un futur proche, une plate-forme supporte l'ajout et la suppression de points de communication, CALICO pourra alors exploiter directement ces fonctionnalités.

D'autre part, le métamodèle de mise à jour est aussi rendu accessible aux outils d'analyse des interactions de CALICO. Ce mécanisme autorise ainsi l'intégration des outils d'analyse incrémentale, comme l'outil de SafArchie. De plus, en fournissant un métamodèle de mise à jour très précis, cela permet aux analyses d'avoir une connaissance détaillée des modifications structurelles et donc potentiellement d'améliorer leur algorithme de composition incrémentale.

Globalement, le métamodèle de mise à jour est indépendant de toute plate-forme. En conséquence, il ne définit aucune contrainte sur l'ordre d'exécution des opérations. Cependant, cet ordre est très important et dépend des contraintes de l'API de déploiement des plates-formes. De plus, certaines plates-formes peuvent ne pas fournir d'opérations de reconfiguration correspondant aux opérations de construction du métamodèle de mise à jour. C'est pour cela que l'étape suivante consiste à réordonnancer et restructurer les opérations de construction du système en fonction de la plate-forme cible, afin de satisfaire les contraintes de cette dernière.

7.4.2 Restructuration des opérations du modèle de mise à jour

La seconde étape du déploiement est l'étape d'ajustement du modèle de mise à jour. Elle consiste à réordonnancer et restructurer les opérations de construction de l'application du modèle de mise à jour afin qu'elles respectent les contraintes de déploiement de la plate-forme cible.

Afin d'être le plus générique possible, la seconde étape repose sur le style architectural *"pipe and filter"*. La figure 7.9 représente schématiquement l'organisation interne de l'étape d'ajustement. Durant cette étape, le modèle de mise à jour est modifié en traversant successivement différents filtres. Un filtre prend en charge la résolution d'un type de contrainte de déploiement, comme par exemple : la plate-forme n'est pas capable d'ajouter et de supprimer des points de communication. Une partie de l'action du filtre correspond à une transformation de modèle. Le filtre prend en entrée un modèle de mise à jour et génère en sortie un nouveau modèle de mise à jour. Ce nouveau modèle satisfait la contrainte de déploiement prise en charge par le filtre. La seconde partie de l'action du filtre est très diverse. Par exemple, elle peut générer le code de nouveaux composants ou de nouveaux services, ou même modifier le code existant à l'aide d'une approche orientée aspect.

Globalement, ce mécanisme permet à CALICO d'être extensible. Il peut s'adapter aux différentes contraintes des plates-formes en ajoutant ou supprimant des filtres dans l'architecture interne de l'étape d'ajustement. De plus, généralement différentes plates-formes peuvent partager en commun des types de contraintes de déploiement similaires. Ce cas est pris en compte dans CALICO car l'architecture *"pipe and filter"* autorise la factorisation des transformations communes grâce à la réutilisation des filtres pour différentes plates-formes.

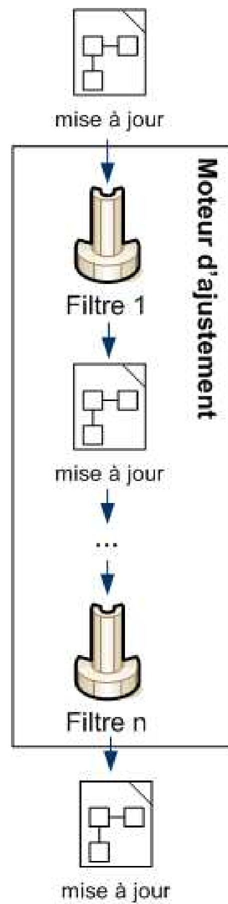

FIG. 7.9: Architecture interne de l'étape d'ajustement

Dans la suite de la section, nous présentons d'abord quelques exemples de filtres. Ensuite, nous illustrons l'utilisation de ces filtres pour élaborer, respectivement, des modèles de mise à jour conformes aux contraintes de déploiement des plates-formes OpenCOM, OpenCCM, Fractal, FraSCAti et WEBservice.

A) Exemples de filtres

Cette section donne cinq exemples de filtres gérant respectivement les problèmes de déploiement liés à l'ajout/suppression de points de communication et de paramètres, au déploiement des composants, au déploiement des services et au tissage des aspects.

Ajout/suppression des points de communication. La très grande majorité des plates-formes à composants et des plates-formes orientées services ne sont pas capables d'ajouter et de supprimer des points de communication pendant l'exécution du système. Le filtre `CommunicationPointConstraint` gère ce type de contrainte. L'action du filtre repose uniquement sur une réécriture de la liste des opérations de construction. Au lieu d'ajouter ou de supprimer un point de communication, ce filtre spécifie une suppression de l'ancienne entité, puis l'ajout de la nouvelle entité. Pour réaliser cet objectif, le filtre utilise l'algorithme suivant : quand le filtre trouve une opération d'ajout ou de suppression de points de communication d'une entité E, il supprime de la liste la totalité des opérations d'ajout/suppression de points de communication de E. Puis il ajoute une opération de suppression de l'entité E, suivie d'une opération d'ajout de l'entité E.

```
UpdateRules CommunicationPointConstraint(UpdateRules r)
{
  Pour chaque Update u de r
  {
    si(u est de type AddCommunication ou RemoveCommunicationPoint)
    {
      Entity E = u.elt.entity;
      supprimer tous les AddCommunication a et RemoveCommunicationPoint b de r tel que
        a.elt.entity==E ou b.elt.entity==E;
      ajouter RemoveEntity(u.elt.entity) dans r;
      ajouter AddEntity(u.elt.entity) dans r;
    }
  }
  retourner r;
}
```

Ajout/suppression de paramètres. De la même manière, la très grande majorité des plates-formes à composants et des plates-formes orientées services ne sont pas capables d'ajouter et de supprimer des paramètres à un composant ou à un service pendant l'exécution du système. Le filtre `ParameterConstraint` gère ce type en contrainte en remplaçant les opérations d'ajout/suppression de paramètre par des opérations de repliement et de déploiement de l'entité concernée.

```
UpdateRules ParameterConstraint(UpdateRules r)
{
  Pour chaque Update u de r
  {
    si(u est de type AddParameter ou RemoveParameter)
    {
      Entity E = u.elt.entity;
      supprimer tous les AddParameter a et RemoveParameter b de r tel que
        a.elt.entity==E ou b.elt.entity==E;
      ajouter RemoveEntity(u.elt.entity) dans r;
      ajouter AddEntity(u.elt.entity) dans r;
    }
  }
  retourner r;
}
```

Ordre de déploiement des composants. Les plates-formes à composants imposent un ordre de déploiement précis. Les sous composants d'un composite doivent être déployés après le composite. Les connecteurs entre composants doivent être créés après le déploiement des composants impliqués dans la connexion. Les aspects doivent être tissés après le déploiement de l'entité concernée par le tissage. Une entité ne peut être supprimée qu'après la déconnexion de cette entité avec toutes les autres entités.

Ce type de contrainte de déploiement est géré par le filtre `ComponentOrder`. Ce filtre réordonnance la liste des opérations de construction sans les modifier. Pour ce faire il place dans la liste les opérations dans l'ordre suivant : les opérations de suppression des aspects, les opérations de suppression des paramètres, les opérations de suppression des connexions, les opérations de suppression des entités en commençant par les entités de plus bas niveau hiérarchique et en remontant récursivement dans la hiérarchie, puis les opérations d'ajout d'entité en commençant par les entités de plus haut niveau hiérarchique et en descendant récursivement dans la hiérarchie, suivie des opérations d'ajout de connexion, des opérations d'ajout de paramètres et des opérations de tissage d'aspect.

```
UpdateRules ComponentOrder(UpdateRules r)
{
  créer UpdateRules res;
  pour chaque Update u de r
    ajouter u dans res si u est de type RemoveInterceptor;
  pour chaque Update u de r
    ajouter u dans res si u est de type RemoveParameter;
  pour chaque Update u de r
    ajouter u dans res si u est de type RemoveRole;
  pour chaque Update u de r
    ajouter u dans res si u est de type RemoveConnector;
  pour chaque Update u de r
    ajouter u dans res si u est de type RemoveEntity et RemoveEntity(u.elt.subEntities) n'est pas dans r;
```

```
  pour chaque Update u de r
    ajouter u dans res si u est de type AddEntity et AddEntity(u.elt.superEntities) n'est pas dans r;
  pour chaque Update u de r
    ajouter u dans res si u est de type addConnector;
  pour chaque Update u de r
    ajouter u dans res si u est de type AddRole;
  pour chaque Update u de r
    ajouter u dans res si u est de type AddParameter;
  pour chaque Update u de r
    ajouter u dans res si u est de type AddInterceptor;
  retourner res;
}
```

Services WEB. Les plates-formes d'exécution des services WEB imposent aussi des contraintes de déploiement qui sont différentes de celles des plates-formes à composants. Dans quelques plates-formes orientées services industrielles, comme GlassFish [SUN09], l'action de connexion entre deux services n'est pas découplée du service. En effet, les références vers les services requis sont spécifiées à l'intérieur du service. Les opérations de déploiement sont donc limitées à deux actions : déploiement et repliement d'un service. Ce mécanisme engendre plusieurs contraintes : d'une part, un service doit être déployé après avoir déployé tous les services requis. D'autre part, la modification d'une connexion entre deux services nécessite de replier le service et de le redéployer avec la nouvelle référence vers le service requis.

Le filtre `WEBServiceOrder` résout ce type de contrainte. Afin de gérer ces contraintes de déploiement des services, le filtre utilise un algorithme composé de deux étapes. La première étape correspond à une transformation du modèle de mise à jour : soit in la liste des opérations de construction en entrée et soit out la liste des opérations de construction en sortie, qui est initialement vide. D'abord, le filtre restructure la liste in telle que : pour chaque opération R de suppression d'une connexion d'un service e vers un service f, le filtre supprime l'opération R de la liste in. Puis, il ajoute dans la liste in les opérations de repliement du service e, puis du déploiement du service e, si et seulement si ces opérations n'existent pas déjà dans la liste in. Ensuite le filtre remplit la liste out telle que : il ajoute d'abord toutes les opérations de repliement des services de in dans out. Ensuite, pour chaque opération A d'ajout d'entité e dans la liste in, il ajoute l'opération A dans la liste out si et seulement s'il n'existe aucune opération d'ajout de service f dans la liste in telle qu'il existe une opération de connexion de e vers f dans la liste in. Après l'ajout de A dans la liste out, le filtre supprime A de in.

La seconde étape correspond à la mise à jour des références des services. Pour chaque opération d'ajout de connexion d'un service e vers un service f, le filtre modifie le code source de e afin de mettre à jour la référence vers le service f. Pour réaliser cette étape, le filtre utilise une approche orientée aspect qui modifie l'annotation de référence `@WebServiceRef` (cf. tableau 2.3).

```
UpdateRules WEBServiceOrder(UpdateRules in)
{
  créer UpdateRules out;
  pour chaque Update R de in tel que R est de type RemoveConnector
  {
    Entity e = R.elt.requiredRole.entity;
    supprimer R de in;
    ajouter RemoveEntity(e) dans in si la règle n'existe pas déjà;
    ajouter AddEntity(e) dans in si la règle n'existe pas déjà;
  }

  pour chaque Update A de in
  {
    si A est de type RemoveEntity
      ajouter A dans out;
    si A est de type AddEntity
    {
      Entity e = A.elt;
      List<Entity> f = e.external.other.entity tel que e.external.provided==false;
      ajouter A dans out si AddEntity(f) n'est pas dans in;
    }
  }
  retourner out;
 }
```

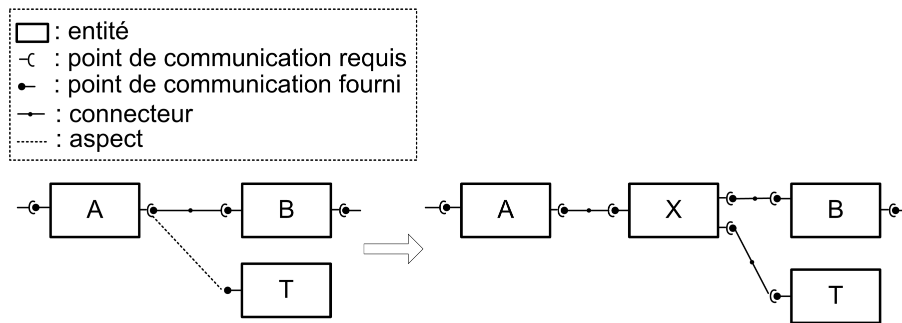

FIG. 7.10: Gestion des intercepteurs

Tissage des aspects. La grande majorité des plates-formes à composants et des plates-formes orientées services ne fournissent pas de fonctionnalité de tissage d'aspect. En effet le tissage d'aspect repose sur un mécanisme d'interception des messages envoyés et reçus par les points de communication. Donc, nous proposons le filtre `AspectSupport` qui fournit un support générique pour créer et insérer des intercepteurs, indépendamment de la plate-forme sous jacente. Un intercepteur `X` est concrétisé sous la forme d'une entité `X`, qui est placée entre les deux entités métiers `A` et `B` interagissant, comme le montre la figure 7.10. L'entité intercepteur `X` possède une paire de points de communication requis/fournis pour propager l'appel de `A` vers `B`. Il possède un troisième point de communication requis qui est connecté avec l'entité `T` codant la préoccupation transverse. Pour ajouter cet intercepteur, le filtre `AspectSupport` remplace toutes les opérations de tissage par des opérations de déploiement de l'entité `X`, suppression des connexions entre `A` et `B`, puis ajout des nouvelles connexions entre `A` et `X`, entre `X` et `T` et entre `X` et `B`. Le filtre utilise l'algorithme inverse pour supprimer les aspects. Le filtre gère aussi la génération du code de l'entité `X`. Concrètement, il utilise le métamodèle de spécification des propriétés comportementales afin de spécifier le comportement de l'entité `X`. Ensuite, il appelle l'outil de génération de code pour générer le composant ou le service correspondant à `X`, en fonction de la plate-forme cible.

```
UpdateRules AspectSupport(UpdateRules in)
{
  pour chaque Update R de in tel que R est de type AddInterceptor
  {
    Aspect e = R.elt;
    CommunicationPoint o = R.in;
    CommunicationPoint p = o.other;
    supprimer R de in;
    ajouter AddEntity(X) dans in;
    ajouter RemoveConnector(k) dans in tel que k.requiredRole==o et k.providedRole==p
                                        ou que k.requiredRole==p et k.providedRole==o;
    ajouter AddConnector(A-X);
    ajouter AddConnector(X-T);
    ajouter AddConnector(X-B);
  }
  retourner in;
 }
```

B) Exemple d'organisation interne

La figure 7.11 présente deux exemples d'organisation interne *"pipe and filter"* des outils de l'étape d'ajustement. Le premier exemple satisfait les contraintes de déploiement communes de la majorité des plates-formes à composants, comme les plates-formes Fractal, OpenCOM, OpenCCM et FraSCAti. Le modèle de mise à jour passe d'abord dans les filtres `CommunicationPointConstraint` et `ParameterConstraint` afin de pallier le manque de support d'ajout/suppression de ports et de paramètres. Ensuite, le modèle passe dans le filtre `AspectSupport` afin de tisser les aspects. Enfin, il entre dans le filtre `ComponentOrder` pour que le modèle satisfasse les contraintes liées à l'ordre de déploiement des plates-formes à composants.

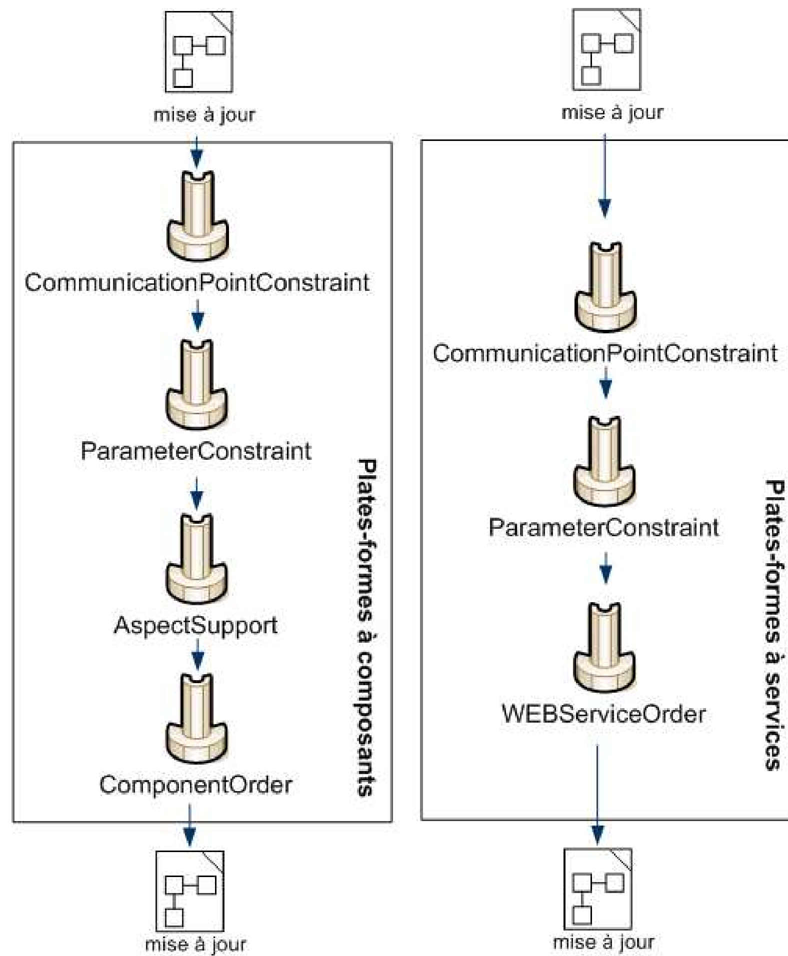

FIG. 7.11: Exemple d'organisation interne

Le second exemple d'organisation satisfait les contraintes de déploiement des services WEB. Le modèle de mise à jour passe successivement dans les filtres `CommunicationPointConstraint`, `ParameterConstraint` et `WEBServiceOrder`.

Exemple. Pour rappel, l'objectif de notre scénario est de propager l'évolution effectuée dans la seconde version du système DMP, c'est-à-dire l'ajout de l'entité `DataConverter`. La comparaison des modèles a produit le modèle de mise à jour suivant :

```
AddConnector between DataConverter.out and PDAClient.searchData
AddConnector between IHM.out and DataConverter.in
AddEntity DataConverter in PDAClient
RemoveConnector between IHM.out and PDAClient.searchData
```

Afin de restructurer ce modèle pour l'adapter aux contraintes de déploiement de la plate-forme Fractal, ce modèle entre successivement dans les filtres `Communication PointConstraint`, `ParameterConstraint`, `AspectSupport` et `ComponentOrder`. Le modèle de mise à jour reste inchangé suite à sa traversée des trois premiers filtres dans la mesure où ce modèle respecte déjà les contraintes liées à l'ajout/suppression des points de communication, des paramètres et des aspects. Néanmoins, ce modèle ne respecte pas la contrainte liée à l'ordre de déploiement des éléments architecturaux. Par exemple, il spécifie un ajout de connecteur entre les entités `PDAClient` et `DataConverter` alors que l'entité `DataConverter` n'est

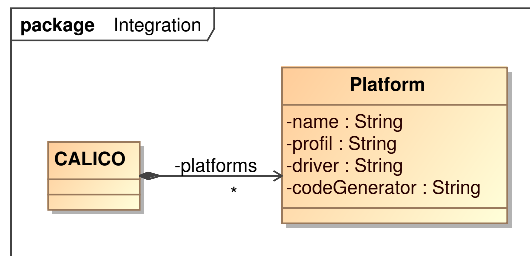

FIG. 7.12: Métamodèle d'intégration

pas encore déployée. Le modèle est donc restructuré par le filtre `ComponentOrder` qui produit le modèle suivant qui est conforme avec les contraintes de déploiement de la plate-forme Fractal :

```
RemoveConnector between IHM.out and PDAClient.searchData
AddEntity DataConverter in PDAClient
AddConnector between DataConverter.out and PDAClient.searchData
AddConnector between IHM.out and DataConverter.in
```

7.4.3 Interprétation du modèle de mise à jour

La troisième et dernière étape du déploiement correspond à l'interprétation des opérations de construction du modèle de mise à jour. Pour réaliser cette étape, CALICO charge un driver en fonction de la plate-forme cible. CALICO se base sur le modèle d'intégration afin de récupérer la classe d'implémentation du driver qui est spécifiée dans la méta-classe `Platform` (cf. figure 7.12). La méta-classe `Platform` contient aussi la liste des filtres d'ajustement.

Un driver est spécifique à une plate-forme. Son rôle est d'invoquer les APIs de déploiement de la plate-forme en fonction des opérations de construction du modèle de mise à jour. Le driver garde aussi le lien entre les éléments d'exécution de la plate-forme, c'est-à-dire les composants ou les services en cours d'exécution, et les concepts du modèle de structure du système.

Pour commencer le déploiement, CALICO appelle la méthode `stop` du driver qui a pour rôle de préparer une adaptation dynamique. Par exemple la méthode peut arrêter le système si la plate-forme le supporte. Ensuite, CALICO utilise le driver pour interpréter le modèle de mise à jour. Grâce à la préparation du modèle de mise à jour dans l'étape d'ajustement, l'étape d'interprétation du modèle est directe. En effet, chaque opération de construction est associée à un bloc d'instruction de l'API de la plate-forme. Concrètement, le driver appelle les APIs de déploiement spécifique à la plate-forme en fonction de l'opération de construction. Le tableau 7.1 donne les exemples d'interprétation des opérations pour les plates-formes OpenCOM, Fractal, OpenCCM et service WEB. Pour finir, CALICO appelle la méthode `start` du driver qui a pour rôle de relancer le système.

Exemple. La dernière étape du déploiement, dans notre scénario, correspond à interpréter le modèle de mise à jour suivant pour la plate-forme Fractal :

```
RemoveConnector between IHM.out and PDAClient.searchData
AddEntity DataConverter in PDAClient
AddConnector between DataConverter.out and PDAClient.searchData
AddConnector between IHM.out and DataConverter.in
```

L'interprétation consiste à invoquer la liste des opérations de déploiement suivantes de la plate-forme Fractal (cf. tableau 7.1) :

Interpretation	OpenCOM	Fractal	OpenCCM	service WEB
AddParameter	n/a	n/a	n/a	n/a
RemoveParameter	n/a	n/a	n/a	n/a
AddEntity	createInstance	newFcInstance addFcSubComponent	create_component	asadmin deploy
RemoveEntity	deleteInstance	removeFcSubComponent	remove	asadmin undeploy
ReplaceEntity	n/a	n/a	n/a	n/a
AddInterceptor	n/a	n/a	n/a	intercpetur glassfich
RemoveInterceptor	n/a	n/a	n/a	intercpetur glassfich
AddConnector	connect	bindFc	_request("connect_"...) _request("subscribe_"...)	n/a
RemoveConnector	disconnect	unbindFc	_request("disconnect_"...) _request("unsubscribe_"...)	n/a
AddRole	connect	bindFc	_request("connect_"...) _request("subscribe_"...)	n/a
RemoveRole	disconnect	unbindFc	_request("disconnect_"...) _request("unsubscribe_"...)	n/a
AddCommunicationPoint	n/a	n/a	n/a	n/a
RemoveCommunicationPoint	n/a	n/a	n/a	n/a
start	startup	startFc	n/a	asadmin enable
stop	shutdown	stopFc	n/a	asadmin disable

TAB. 7.1: Interprétation du modèle de mise à jour

```
DMP.stopFc()
IHM.unbindFc("out");
DataConverter = newFcInstance(...);
PDAClient.addFcSubComponent(DataConverter);
DataConverter.bindFc("out",PDAClient.searchData);
IHM.bindFc("out",DataConverter.in);
DMP.startFc();
```

7.4.4 Discussion

L'organisation de l'étape de déploiement en trois sous étapes permet à CALICO d'être extensible et multi plates-formes. De cette manière, les algorithmes génériques sont clairement factorisés et isolés des algorithmes spécifiques à chaque plate-forme. L'étape de comparaison des modèles permet de gérer sans distinction les évolutions qui correspondent à des ajouts et des suppressions de fonctionnalités, indépendamment de la plate-forme cible. L'étape d'ajustement factorise les algorithmes complexes de résolution des contraintes de déploiement communs et les isole dans des filtres séparés. Ces filtres peuvent être réutilisés pour les différentes plates-formes qui partagent des contraintes de déploiement identiques. Le driver est, quant à lui, spécifique à une plate-forme. Il encapsule l'interprétation du modèle de mise à jour vers l'API d'une plate-forme. L'interprétation est directe car le modèle de mise à jour a été restructuré au préalable durant l'étape d'ajustement.

Il existe différents outils de déploiement, comme Fscript [PMSD07]. Fscript implémente un algorithme de déploiement complexe qui prend en charge le déploiement transactionnel, c'est-à-dire que si le déploiement échoue, le système n'est pas laissé dans un état instable, mais il est remis dans la configuration initiale. Dans, CALICO, notre objectif a été de produire un canevas de développement le plus extensible et générique possible afin de pouvoir exploiter les travaux existants. Nous avons aussi appliqué cette démarche pour la conception de l'étape de déploiement de CALICO. Il est donc ainsi possible de créer un driver basé sur Fscript et d'exploiter le support du déploiement transactionnel de l'outil.

Le projet FAROS a pour objectif de permettre l'expression de contraintes applicatives indépendamment de la plate-forme cible, puis de projeter ces contraintes vers une plate-forme cible. Pour réaliser cet objectif, le projet FAROS a décidé d'utiliser deux métamodèles. Le métamodèle pivot représente une architecture indépendamment de la plate-forme, il a le même rôle que les métamodèles de CALICO. Le métamodèle de plate-forme décrit une plate-forme donnée. Avec ce principe, la propagation des contrats correspond à une transformation de

modèle, qui prend en entrée un modèle pivot et produit en sortie un modèle plate-forme. Dans CALICO nous avons choisi de ne pas utiliser de métamodèle de plate-forme, car CALICO est une approche dynamique. En effet, un métamodèle de plate-forme ne permet de décrire que l'état de la plate-forme à un instant donné. C'est une approche statique. Dans CALICO, nous avons besoin d'exprimer l'ordre des modifications car les plates-formes possèdent des contraintes de déploiement. Donc au lieu de définir un métamodèle de plate-forme pour chaque plate-forme, nous décrivons les opérations élémentaires de déploiement à exécuter pour propager les changements dans la plate-forme. De plus, comme l'ensemble des opérations de déploiement est commun à différentes plates-formes, cela permet à CALICO d'être plus facilement extensible que FAROS. Le support d'une nouvelle plate-forme consiste à définir un nouveau driver qui interprète ces opérations de déploiement en invoquant l'API de la plate-forme, alors que dans le cas de FAROS, il faut redéfinir un nouveau métamodèle de plate-forme et l'algorithme de transformation.

7.5 Étape de validation dynamique

Une fois que le système est déployé sur la plate-forme d'exécution, la dernière étape du cycle de développement correspond à la validation dynamique du système par les testeurs. Pendant cette étape, les testeurs exécutent les scénarios d'utilisation du système afin de vérifier qu'aucune interaction partiellement compatible n'ait violée. Classiquement, lorsqu'une erreur est détectée, le testeur reçoit le numéro de ligne de l'implémentation du composant ou du service où l'erreur a été émise, ainsi que la pile d'appel. Ces messages d'erreur sont donc de très bas niveau et ne sont compréhensibles que par les développeurs. Il est donc nécessaire que le développeur communique avec l'architecte afin d'identifier les propriétés applicatives qui ont été violées et le problème de conception qui a entraîné cette erreur.

Cette tâche est automatiquement effectuée par l'outil de débogage de CALICO. Cet outil est situé dans le niveau modèle et est indépendant de toute plate-forme. Il implémente un moteur de règle. En fonction des événements reçus provenant du système en cours d'exécution, l'outil parcourt les analyses dynamiques spécifiées dans le modèle de débogage et déclenche l'exécution de l'analyse dynamique appropriée. Lorsque l'outil détecte une erreur, il signale directement à l'architecte le problème en lui indiquant la propriété applicative qui a été violée. Pour identifier la propriété qui est violée, chaque analyse dynamique `Rule` est associée avec le contrat qui est dynamiquement validé, comme le montre la figure 7.13. Concrètement, chaque fois que l'outil d'analyse des interactions ajoute une nouvelle analyse dynamique, il spécifie la propriété applicative impliquée, afin de garder le lien causal entre le contrat et le mécanisme d'analyse dynamique associé. Ainsi, l'outil de débogage peut produire des messages d'erreur directement en lien avec la conception.

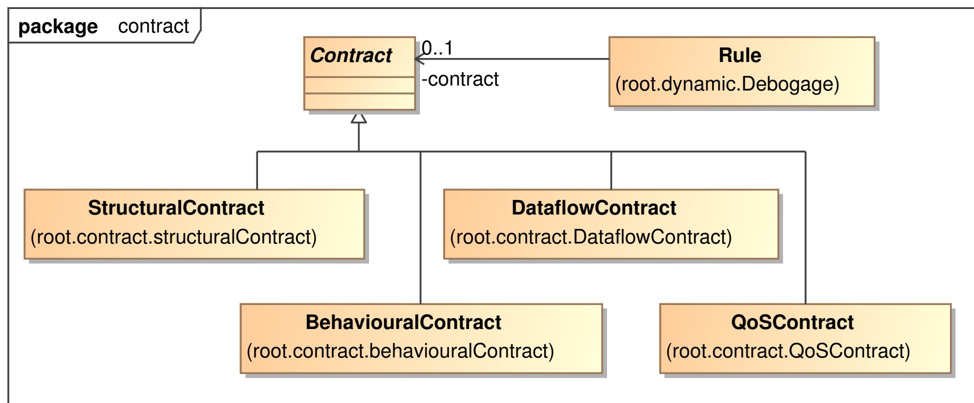

FIG. 7.13: Métamodèle des spécifications contractuelles

Exemple. Une fois que la seconde version de l'architecture DMP est déployée, les testeurs peuvent poursuivre l'exécution des scénarios d'utilisation. Avec cette seconde version, le radiologue peut consulter des radiographies sur son PDA dans la mesure où le filtre de conversion d'image `DataConverter` réduit leur taille. Néanmoins, cette seconde version n'a pas apporté de solution au fait que le PDA ne peut recevoir que des images de type JPG et du texte ASCII. La validation dynamique va donc consister à vérifier si le PDA reçoit des images qui ne sont pas de type JPG. Alors, chaque fois que le serveur `MedicalServer` retourne une donnée médicale à l'entité `GlobalSearch`, l'intercepteur envoie le type de la donnée à l'outil de débogage. L'outil de débogage exécute l'analyse dynamique qui correspond à l'événement d'envoi de messages. Cette analyse teste si la donnée est de type JPG. Si l'analyse déclenche une erreur, elle signale à l'architecte que la spécification de flot de données `D1` est violée. De cette manière, l'architecte peut, par exemple, modifier son architecture en ajoutant, dans l'itération de développement suivante, une entité qui convertit une image en un type JPG entre les entités `GlobalSearch` et `MedicalServer`. Ensuite, si l'outil d'analyse des interactions ne détecte aucune interaction incompatible, les modifications sont propagées dans le système en cours d'exécution afin de revalider dynamiquement le système.

7.6 Conclusion

Nous avons présenté les outils du support d'exécution de CALICO permettant un couplage fort entre les cinq étapes du cycle de développement, c'est-à-dire les étapes de conception, d'implémentation, d'instrumentation, de déploiement et de validation dynamique. Ce couplage fort permet à CALICO d'avoir un contrôle fin sur la globalité du cycle de développement du système. Il garantit une forte synchronisation entre la description de la conception du système et le système en cours d'exécution. Ainsi, ce support prend en charge, sans distinction, les évolutions effectuées par l'architecte, qu'elles correspondent à des ajouts ou des suppressions de fonctionnalités. De plus, il empêche l'introduction d'incohérence lors du passage entre les étapes du cycle de développement.

En outre, CALICO facilite aussi la tâche des différents acteurs durant le développement du système. En effet, les développeurs ont uniquement besoin de se focaliser sur le code métier dans la mesure où tout le code technique est généré par CALICO et est constamment gardé synchronisé avec la conception. Mais aussi, CALICO est capable d'instrumenter automatiquement un système afin que celui-ci puisse observer les informations d'exécution requises par les analyses dynamiques. De la même manière, la tâche de l'administrateur système est aussi facilitée car CALICO gère complètement le déploiement de l'application de manière incrémentale. Mais encore, la validation dynamique du système est aussi simplifiée dans la mesure où la mise en œuvre des analyses dynamiques est entièrement pris en charge par CALICO, et que les erreurs sont directement remontées au niveau modèle et sont compréhensibles par l'architecte logiciel. Ainsi, l'architecte peut, à n'importe quel moment, ajuster sa conception pour corriger le problème dans l'itération suivante du cycle de développement.

CALICO permet aussi un support précis lié à la validation dynamique. Le but étant de limiter le surcoût dû à l'exécution d'actions inutiles. Par exemple, le modèle de mise à jour qui décrit les modifications structurelles, est à granularité extrêmement fine. Il est capable d'exprimer des modifications de la structure d'une entité. Ces informations supplémentaires sont utilisées par l'outil de déploiement afin que ce dernier n'effectue que les actions de déploiement strictement nécessaires pour synchroniser le système en cours d'exécution. De la même manière, CALICO a la capacité d'ajouter et de supprimer des sondes du système afin de ne capturer exclusivement que les événements requis pour finaliser la validation des interactions partiellement compatibles. Cela évite un surcoût lié à l'observation d'événements inutiles.

D'autre part, nous avons conçu les outils du support d'exécution toujours avec l'optique de produire un canevas de développement générique et extensible. CALICO autorise la prise en compte de nouvelles plates-formes et de nouvelles sondes de QdS. Les outils du support d'exécution de CALICO ont été conçus afin de faciliter le support de nouvelles plates-formes.

Ils ne reposent pas sur des mécanismes spécifiques à une plate-forme. De plus, les algorithmes génériques sont clairement isolés des algorithmes spécifiques à chaque plate-forme afin de permettre leurs réutilisations. Cette extensibilité repose sur le modèle d'intégration qui permet de décrire de manière uniforme l'architecture interne de CALICO et les relations entre les différents outils de CALICO. De cette manière, un expert du domaine peut ajouter le support de nouvelles plates-formes et de nouvelles sondes de QdS en modifiant le modèle d'intégration. Ceci est même possible pendant l'utilisation de CALICO pour développer un système.

Globalement, CALICO permet d'effectuer une validation dynamique d'un système même si la plate-forme d'exécution sous jacente ne fournit aucun mécanisme d'analyse dynamique. La plate-forme a uniquement besoin de gérer les adaptations dynamiques, ce qui est une fonctionnalité courante des plates-formes récentes.

Chapitre 8

Concrétisation et évaluation de CALICO

Sommaire

8.1 Implémentation de CALICO **153**
- 8.1.1 Implémentation des métamodèles de CALICO 154
- 8.1.2 Implémentation de l'outillage de CALICO 160
- 8.1.3 Bilan 165

8.2 Évaluation de CALICO **165**
- 8.2.1 Extensibilité 166
- 8.2.2 Performance 168
- 8.2.3 Apport aux acteurs du développement 173

8.3 Conclusion et perspectives **174**

Ce chapitre présente la dernière contribution de cette thèse qui concerne la concrétisation de CALICO. L'objectif est de proposer un environnement de développement intégré (IDE pour "*Integrated Development Environment*") pour la conception et l'évolution agile de système à composants ou orientés services. Cette implémentation permet de concrétiser l'ensemble des propositions de CALICO faites dans cette thèse. Elle sert aussi de base pour évaluer qualitativement et quantitativement notre approche. La section 8.1 détaille l'implémentation du canevas CALICO élaboré dans le cadre de cette thèse. Elle fournit des informations quantitatives de cette implémentation. La section 8.2 décrit les évaluations qualitative et quantitative que nous avons effectuées concernant l'extensibilité et la performance de notre implémentation.

8.1 Implémentation de CALICO

Pour implémenter CALICO, nous avons décidé de nous baser sur l'environnement de développement intégré Eclipse [1]. Eclipse est un IDE libre, multi-langage et extensible. Il est développé activement par la fondation Eclipse qui regroupe des grands industriels, comme par exemple IBM, Nokia, Oracle et Motorola. Il est bien connu des développeurs, et est largement utilisé, à la fois dans la communauté du logiciel libre et dans le monde industriel.

La spécificité d'Eclipse provient de son découpage en plugins. Chaque fonctionnalité d'Eclipse est ainsi isolée dans des plugins séparés. De plus, Eclipse propose un ensemble de librairies permettant à des développeurs tiers de créer leurs propres extensions. Nous avons donc décidé de nous baser sur Eclipse et de développer CALICO en Java.

[1] http ://www.eclipse.org

L'implémentation de CALICO est composée de deux parties : la partie métamodèle et la partie outillage. L'implémentation de la partie métamodèle est présentée dans la section 8.1.1. Celle des outils de CALICO est décrite dans la section 8.1.2. Enfin, la section 8.1.3 donne le bilan de l'implémentation de CALICO.

8.1.1 Implémentation des métamodèles de CALICO

L'implémentation des métamodèles de CALICO inclut deux parties. La première partie concerne l'implémentation des métamodèles en eux-mêmes, c'est-à-dire les métamodèles de la structure du système, des contrats, de débogage, d'aspect et de mise à jour. La seconde partie correspond à l'implémentation d'un modeleur graphique permettant à l'architecte logiciel de concevoir et de corriger son système.

A) Mise en œuvre des métamodèles

Afin d'implémenter les métamodèles de CALICO, nous avons besoin d'un support qui permet l'écriture directe des métamodèles et qui propose une démarche d'ingénierie dirigée par les modèles. Nous avons donc choisi de nous reposer sur l'IDE Eclipse qui fournit le canevas EMF [2] (EMF pour *"Eclipse Modeling Framework"*).

Présentation d'EMF. Le canevas Eclipse EMF est un canevas de modélisation et de génération de code pour la construction d'applications reposant sur une démarche d'ingénierie dirigée par les modèles. EMF permet, à partir d'une description sous forme de métamodèles, de générer le code Java correspondant à l'implémentation des métamodèles. EMF produit aussi l'ensemble des outils pour créer et manipuler les modèles conformes aux métamodèles.

Comme EMF repose sur une démarche d'ingénierie dirigée par les modèles, l'implémentation d'un métamodèle correspond à instancier un métamétamodèle. Le métamétamodèle est appelé un métamétamodèle Ecore. Le métamétamodèle Ecore, représenté dans la figure 8.1, est aligné avec le standard eMOF de l'OMG [Obj03b]. Le métamétamodèle permet de définir des classes (`EClass`) nommées. Une classe possède zéro ou plusieurs attributs (`EAttribute`) nommés et typés (`EDataType`). Une classe possède aussi zéro ou plusieurs références (`EReferences`)

[2] http ://www.eclipse.org/modeling/emf/

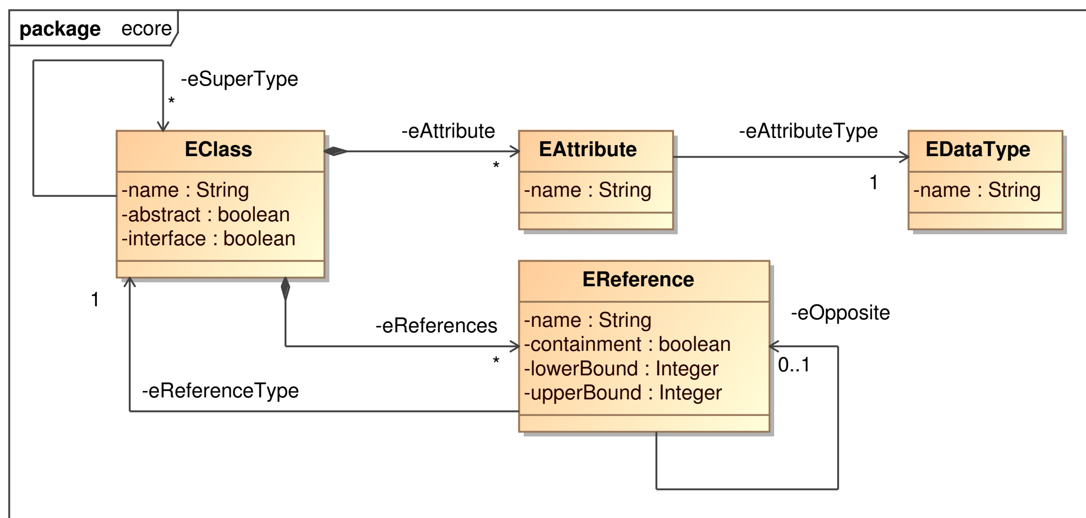

FIG. 8.1: Métamodèle simplifié d'EMF

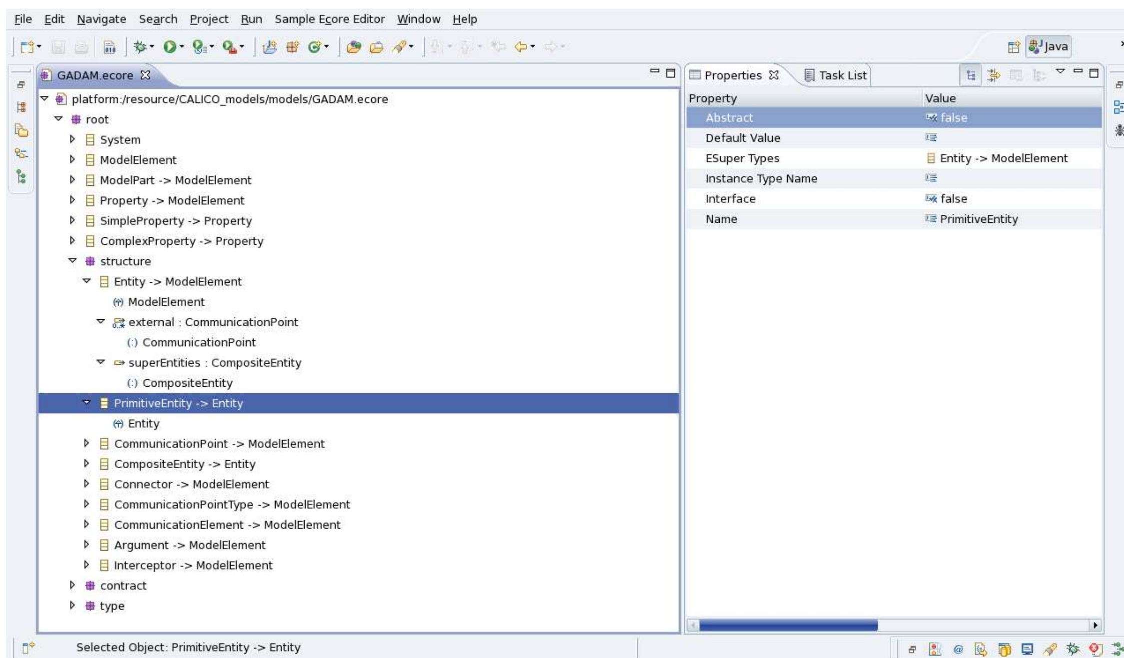

FIG. 8.2: Capture d'écran de l'éditeur arborescent

nommées vers d'autres classes. La métamétaclasse `EReferences` spécifie aussi la cardinalité minimale et maximale de la référence et précise si la référence est une association ou une composition. L'association `eSuperType` permet de définir les relations d'héritage de la classe.

Afin d'éditer les métamodèles, qui sont donc des instances d'Ecore, le canevas EMF propose trois types de moyen. Le premier moyen correspond à écrire manuellement les fichiers textes ecore. Ces fichiers ecore sont des fichiers XML conformes avec le standard XMI, qui est un standard d'échange de métadonnées UML en XML et qui vise à permettre l'interopérabilité entre les différents éditeurs de modèles. Le deuxième moyen est d'utiliser l'outil d'édition arborescent représenté sur la figure 8.2. Le troisième moyen consiste à dessiner graphiquement le métamodèle à l'aide de l'éditeur graphique qui est similaire à un modeleur UML.

Une fois que les métamodèles sont écrits, le canevas EMF fournit une chaîne de processus de génération afin de créer les fichiers sources Java qui correspondent à l'implémentation des métamodèles. La chaîne de processus est représentée sur la figure 8.3. D'abord le fichier de métamodèle ecore est utilisé pour produire un fichier `genmodel`. Ce fichier `genmodel` enrichit la description du métamodèle avec des informations dédiées à la génération de code. Par exemple, il précise le nom du projet Eclipse de destination et définit les paramètres de configuration de l'apparence de l'éditeur arborescent. Pour finir ce fichier `genmodel` sert à produire les fichiers sources Java. Ces fichiers Java générés sont répartis dans trois projets Eclipse distincts. Le premier projet contient le code Java du métamodèle. Le second projet correspond au code du plugin qui prend en charge la manipulation et l'édition des modèles. Le troisième projet contient le code de l'éditeur arborescent dédié à la manipulation des modèles.

Métamodèles de CALICO. Grâce à l'utilisation d'EMF, l'implémentation des métamodèles de CALICO correspond à une écriture directe de nos métamodèles présentés dans les chapitres précédents, en utilisant l'éditeur arborescent (cf. figure 8.2). Le code généré de l'implémentation des métamodèles est contenu dans le projet Eclipse `CALICO_models`. Il permet une manipulation programmatique des modèles de CALICO par l'outil d'analyse et par les outils du support d'exécution. Le code généré permettant la manipulation des modèles est isolé dans le projet Eclipse `CALICO_models.edit` et le code de l'éditeur arborescent est généré dans le projet

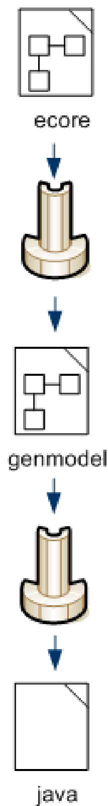

FIG. 8.3: Chaîne de processus de génération d'EMF

CALICO_models.editor.

D'un point de vue quantitatif, l'implémentation des métamodèles correspond à 1150 lignes de XML (dont 647 lignes de ecore et 503 lignes de genmodel). Le processus de génération a produit 50033 lignes de code Java, dont 19498 lignes pour le projet CALICO_models, 9512 lignes pour le projet CALICO_models.edit et 21023 lignes pour le projet CALICO_models.editor.

B) Modeleur graphique

Afin de permettre à un architecte de concevoir son système, nous désirons fournir un modeleur graphique. De plus, nous voulons autoriser différentes représentations graphiques afin que l'architecte puisse choisir la représentation qu'il connaît le mieux. D'autre part, nous souhaitons que le choix de la vue graphique n'ait, évidemment, aucun impact sur les autres parties de l'implémentation. Nous avons donc choisi d'utiliser le canevas GMF [3] (GMF pour *Graphical Modeling Framework*) proposé par l'IDE Eclipse.

Présentation de GMF. Le canevas Eclipse GMF est un canevas de génération et d'exécution d'éditeurs graphiques reposant sur le canevas EMF. Il permet donc de générer un éditeur graphique dédié pour la manipulation d'instance de modèle EMF. GMF repose aussi sur une démarche d'ingénierie dirigée par les modèles. Ainsi, la conception de l'éditeur graphique consiste à créer des instances modèles permettant d'associer une représentation graphique à chaque concept du métamodèle ecore. Il est donc ainsi possible de proposer différentes vues graphiques d'un même métamodèle.

[3]http ://www.eclipse.org/gmf/

La figure 8.4 décrit la chaîne de processus de construction d'un éditeur graphique GMF. La première étape consiste à définir les formes graphiques, comme par exemple les rectangles et les lignes qui représentent graphiquement les concepts du métamodèle. Pour réaliser cette étape, GMF fournit deux métamodèles : `gmftool` et `gmfgraph`. Ces deux métamodèles sont complètement indépendants d'un métamodèle ecore. Cela permet la réutilisation des concepts graphiques pour différents métamodèles. Donc, pour définir les formes graphiques, le développeur a besoin de créer deux modèles conformes aux métamodèles. Le modèle `gmftool` définit les icônes de la barre d'outils d'édition du modèle. Ce sont les icônes que l'on peut voir à droite dans la figure 8.5 qui est une capture d'écran de l'éditeur de CALICO. Le modèle `gmfgraph` spécifie, quant à lui, les figures graphiques utilisées pour la représentation. Il existe deux types deux figures graphiques : les noeuds et les liens. Par exemple, le développeur peut définir des noeuds de forme rectangulaire, ovales, ou même de formes plus complexes, comme la forme des points de communication de CCM. Le développeur peut représenter un lien sous forme de lignes. Ces exemples de représentation sont visibles au centre de la figure 8.5.

La seconde étape consiste à associer chaque figure graphique définie dans le modèle `gmfgraph` avec un concept du métamodèle ecore. Pour réaliser la seconde étape, le développeur a besoin de créer un modèle conforme au métamodèle `gmfmap` (cf. figure 8.4). Avec ce modèle, il associe chaque figure de type de noeud avec un concept de type `EClass` et chaque figure de type lien avec un concept de type `EReference`.

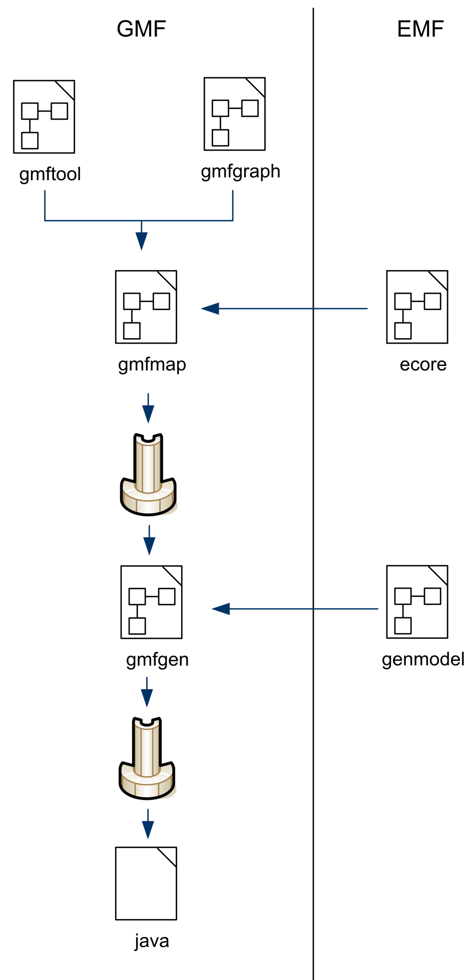

FIG. 8.4: Chaîne de processus de génération de GMF

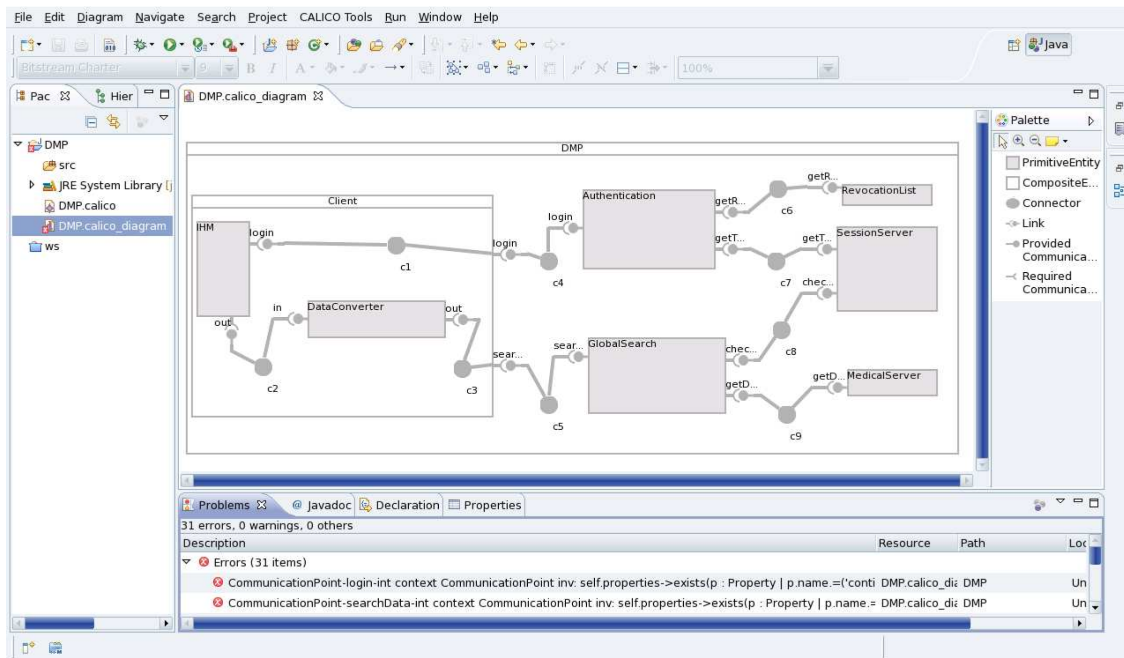

FIG. 8.5: Modeleur graphique structurel de CALICO

Une fois que les associations sont définies, GMF fournit un générateur produisant un modèle `gmfgen`. Ce modèle enrichit le modèle `gmfmap` avec des informations dédiées à la génération de code, comme par exemple le nom du projet Eclipse de destination. Pour finir, à partir de ce modèle `gmfgen` et du modèle `genmodel`, GMF génère le code Java de l'éditeur.

Éditeur graphique de CALICO. Dans le cadre de CALICO, nous avons utilisé GMF pour concevoir un modeleur graphique. Nous avons décidé de créer trois vues différentes : une vue structurelle, une vue comportementale et une vue des points de communication. Pour la vue structurelle, nous avons décidé d'utiliser une forme proche de celle de CCM, comme le montre la figure 8.5. Pour la vue comportementale, nous avons choisi d'utiliser la représentation de BPMN, comme nous pouvons le voir dans la figure 8.6. Enfin pour la vue des points de communication, nous avons opté pour une représentation dédiée (cf. figure 8.7).

Pour chacune de ces trois vues, nous avons créé les modèles `gmftool`, `gmfgraph` et `gmfmap` dans le projet Eclipse `CALICO_GUI`. Nous avons ensuite généré le code Java des éditeurs dans les projets Eclipse `CALICO_GUI.structural.diagram`, `CALICO_GUI.behavioural.diagram` et `CALICO_GUI.commPoint.diagram`. Le tableau 8.1 résume le nombre de lignes de code XML et Java qui ont été écrites et générées pour chacune des vues. Au total, cela correspond à 3822

	vue structurelle	vue comportementale	vue des points de communication	Total
`gmftool`	88	119	81	288
`gmfgraph`	67	134	250	451
`gmfmap`	225	296	213	734
`gmfgen`	720	1026	603	2349
`java`	14183	12371	12895	39449
Total	15283	13946	14042	43271

TAB. 8.1: Nombre de lignes de code pour le modeleur graphique

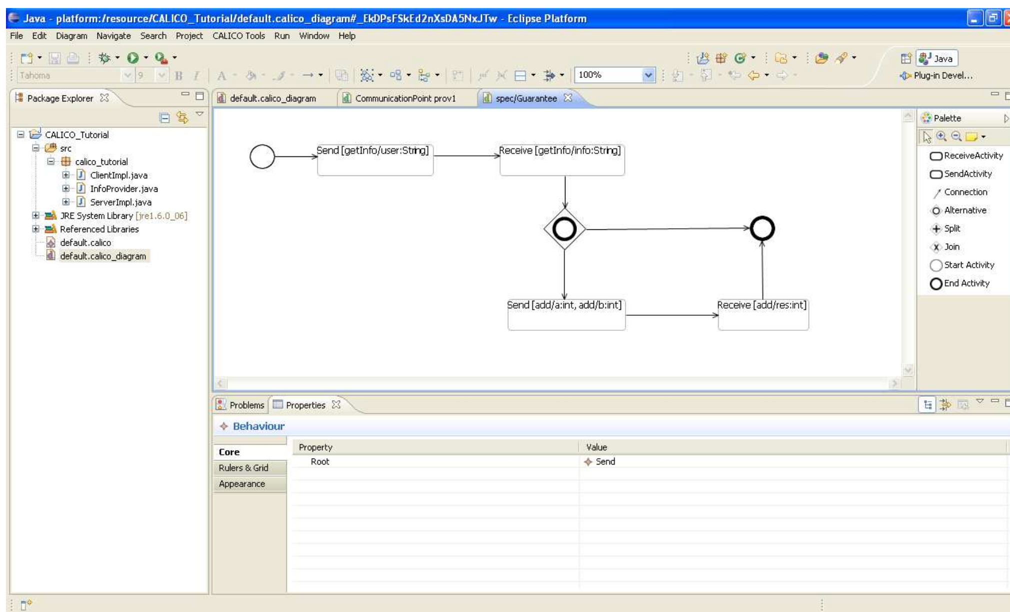

FIG. 8.6: Modeleur graphique comportemental de CALICO

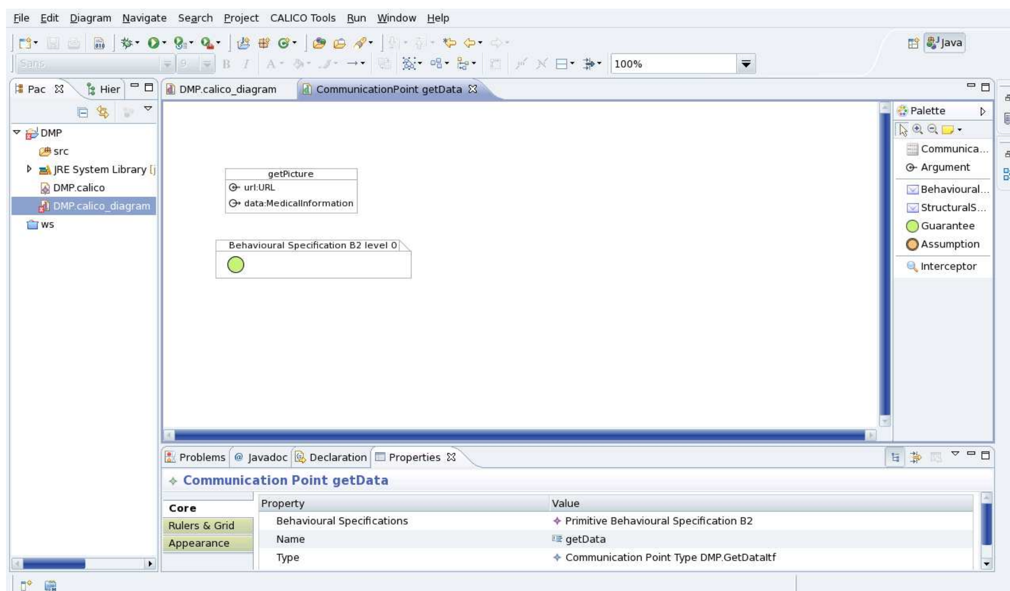

FIG. 8.7: Modeleur graphique des points de communication de CALICO

lignes de XML (dont 2349 généré) et 39449 lignes de Java générées.

Globalement, grâce à l'utilisation de GMF, le modeleur graphique de CALICO est extensible. Il autorise l'ajout de nouvelle vue graphique, comme par exemple une vue structurelle similaire à celle de Fractal ou une vue des points de communication identique aux diagrammes de classes d'UML, en définissant de nouveaux modèles GMF.

8.1.2 Implémentation de l'outillage de CALICO

La seconde partie de l'implémentation correspond à la mise en œuvre des outils de CALICO. Afin d'être modulaire, le code de chaque outil est isolé dans des projets Eclipse distincts. De plus, lors du développement de CALICO, nous avons toujours choisi de proposer une solution extensible, afin qu'il soit possible d'intégrer de nouvelles analyses, de nouvelles plates-formes et de nouvelles sondes. Nous avons donc, à chaque fois, défini une API générique permettant l'extension de CALICO. La suite de cette section présente successivement la mise en œuvre des outils d'analyse des interactions, de génération de code, d'instrumentation, de chargement et de débogage.

A) Outil d'analyse des interactions

L'implémentation de l'outil d'analyse des interactions est contenue dans le projet Eclipse `CALICO_analysisTool`. Cet outil est composé d'une partie générique et d'une partie extensible constituée de plugins.

La partie générique sélectionne les analyses statiques à exécuter en fonction du modèle d'intégration qui décrit les dépendances entre les analyses statiques (cf. figure 6.12). De plus, cette partie utilise le mécanisme de chargement dynamique de classe de Java afin de charger et d'exécuter les analyses statiques de manière transparente. De cette manière, il est possible d'ajouter de nouvelles analyses statiques et de les prendre en compte sans redémarrer CALICO.

Le rôle de la partie extensible est de permettre l'intégration dans CALICO de nouvelles analyses statiques. Cette partie se base sur un mécanisme de plugins. Chaque plugin encapsule un outil d'analyse. Un plugin d'analyse doit implémenter l'interface générique `IAnalyzer` représentée dans la figure 8.8. Cette interface est constituée d'une seule méthode. La méthode `analyzeModel` déclenche l'exécution de la validation des modèles de CALICO. Elle prend en entrée les modèles du système provenant de l'itération précédente (`systemModel`), les modèles du système provenant de l'itération courante du cycle de développement (`designModel`) et le modèle de mise à jour (`update`). Elle produit en sortie l'ensemble des messages d'erreur qui seront transmis à l'architecte logiciel. Afin de fournir un IDE complètement intégré, ces messages d'erreur sont affichés dans la vue d'erreur de l'IDE Eclipse, exactement de la même manière que les erreurs de compilation. La méthode `analyzeModel` modifie aussi les modèles `designModel` pour y intégrer les nouveaux contrats calculés.

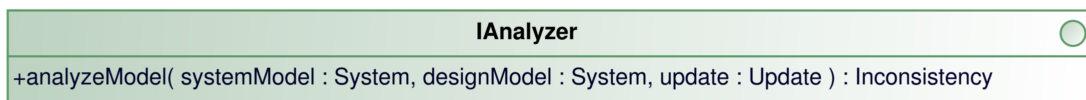

FIG. 8.8: Interface `IAnalyzer`

L'implémentation actuelle de CALICO supporte cinq plugins d'analyse, dont deux plugins pour l'analyse du comportement :

Structure. L'analyse de la structure consiste à valider les invariants OCL. L'implémentation du plugin d'analyse de la structure est directe. Elle utilise la librairie EMF-OCL [4] qui permet la validation de contraintes OCL associées à un modèle EMF.

Comportement. L'implémentation du plugin d'analyse du comportement a consisté à intégrer l'outil SFSP [5]. D'autre part, nous avons aussi développé un second plugin d'analyse du comportement reposant sur les protocoles comportementaux de SOFA. Concrètement, nous avons directement réutilisé et adapté le code de l'analyse du comportement qui a été porté sur Fractal, appelé Fractal-bpc [6], dans CALICO.

[4] http ://www.eclipse.org/modeling/mdt/downloads/ ?project=ocl
[5] http ://transat.gforge.inria.fr
[6] http ://fractal.ow2.org/fractalbpc/index.html

Flot de données. L'implémentation du plugin d'analyse du flot de données repose sur l'outil de calcul symbolique Maxima [7] et Sage [8].

QdS. L'implémentation du plugin d'analyse de QdS repose aussi sur l'outil de calcul symbolique Maxima.

D'un point de vue quantitatif, le tableau 8.2 résume le nombre de lignes de code nécessaires pour développer l'outil d'analyse. Généralement, un plugin est relativement simple et contient peu de lignes de code car il encapsule un outil d'analyse existant. Dans ce cas, le rôle du plugin est de récupérer les informations contenues dans les modèles de CALICO et d'appeler l'outil d'analyse existant. L'encapsulation d'un outil d'analyse existant au sein d'un plugin est plus ou moins directe en fonction du support offert par l'outil. Nous avons identifié trois cas différents. (1) Dans le meilleur des cas, l'outil est une librairie qui fournit une API de programmation. L'écriture du plugin correspond alors à faire la passerelle entre les modèles de CALICO et la librairie en invoquant l'API. (2) Dans le cas où l'outil n'offre pas d'API mais dont le code source est disponible, l'écriture du plugin consiste à modifier une partie du code source de l'outil afin que l'outil utilise en entrée les modèles de CALICO. (3) Dans le pire cas, l'outil n'offre pas d'API et est une boîte noire fermée. Le rôle du plugin est alors d'exécuter l'outil externe dans un nouveau processus et d'analyser la réponse, comme par exemple le texte renvoyé par l'outil.

	partie commune	OCL	SFSP	BPC	flot de données	QdS	Total
Java	395	194	273	283	340	380	1865
Librairies utilisées	n/a	EMF-OCL	SFSP	fractal-bpc	Maxima/Sage	Maxima	n/a

TAB. 8.2: Nombre de lignes de code pour l'outil d'analyse

B) Outil de génération de code

L'outil de génération de code produit le squelette de code et l'ADL conformément à la description de l'architecture, en fonction de la plate-forme cible. Ce code est généré directement dans un projet Eclipse afin de permettre aux développeurs de remplir les squelettes de code avec l'éditeur d'Eclipse, puis de les compiler au sein d'Eclipse. L'implémentation de l'outil de génération de code est composée de deux partie : la partie générique et la partie spécifique à une plate-forme.

La partie générique exporte un ensemble d'API utilitaires facilitant la génération de code Java. Par exemple, elle contient une fonctionnalité, reposant sur le patron de conception visiteur, qui visite le modèle de la structure du système et génère le squelette de code Java des entités. La partie générique est contenue dans le projet Eclipse `CALICO_codeInstrumentation`.

La partie spécifique à une plate-forme exploite les API utilitaires afin de générer le code pour une plate-forme cible. Cette partie est conçue sous forme de plugins chargés dynamiquement en fonction de la plate-forme cible choisie par l'architecte. Un plugin doit étendre la classe abstraite `Model2CodeGeneration` représentée à la figure 8.9. Cette classe implémente l'interface `ICodeManipulation`, qui décrit une manipulation de code source. Cette interface est composée de trois méthodes. La méthode `process` prend en entrée les modèles de CALICO (`system`). Elle déclenche l'exécution de la manipulation de code, c'est-à-dire la génération de code dans le cas présent. L'action de la méthode est configurée à l'aide de propriétés. Dans le cas de la génération de code, il n'y a qu'une seule propriété qui correspond au répertoire de destination dans lequel le code est généré. Le plugin de génération de code spécifique à une plate-forme `X` est contenu dans le projet Eclipse `CALICO_PlateformX` qui regroupe toutes les implémentations dédiées à la plate-forme `X`.

[7] http ://maxima.sourceforge.net
[8] http ://www.sagemath.org

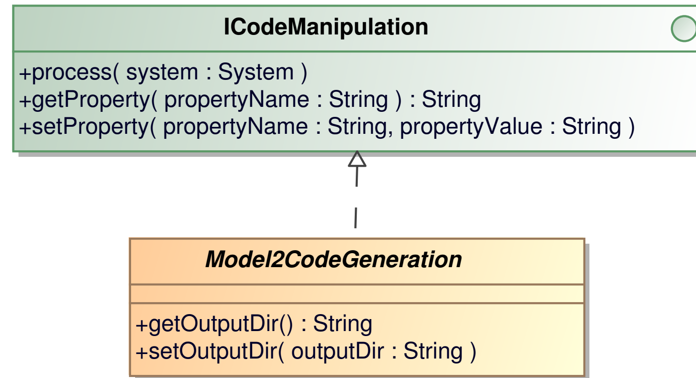

FIG. 8.9: Interface `ICodeManipulation`

La version courante de CALICO prend en charge la génération du squelette de code et de l'ADL de cinq plates-formes, qui sont Fractal, OpenCOM, OpenCCM, FraSCAti et les services WEB. Le tableau 8.3 donne un bilan quantitatif. Il expose le nombre de lignes de code de la partie générique et de la partie spécifique pour chacune des plates-formes prises en charge par CALICO. On peut voir que la plus grande partie du code de génération est factorisée dans la partie générique. Le plugin spécifique à chaque plate-forme est relativement simple et utilise en grande partie l'API de la partie générique. Cependant, nous pouvons constater que le plugin des services WEB contient beaucoup de lignes. Cela s'explique par le fait que ce plugin n'exploite que très peu l'API générique dans la mesure où il ne génère presque pas de fichiers Java. Ce plugin génère un grand nombre de fichiers XML, correspondant aux fichiers de compilation ant, aux fichiers de personnalisation JAXB et JAX-WS, au fichier de déploiement `web.xml`, etc.

	Outil de génération de code	Outil Chargement	Total
Partie commune	1101	3754	4855
Fractal	225	737	962
OpenCOM	63	352	415
OpenCCM	199	409	608
FraSCAti	697	875	1572
WEB service	1018	493	1511

TAB. 8.3: Nombre de lignes de code pour l'outil de génération de code et de chargement

C) Outil d'instrumentation

L'outil d'instrumentation consiste à insérer et supprimer des sondes dans le système en fonction des données d'exécution requises par les analyses dynamiques. Cet outil est aussi composé de deux parties.

La partie générique implémente l'algorithme qui compare les deux versions du modèle de débogage afin d'identifier les nouvelles et les anciennes analyses dynamiques. Ensuite, pour chacune des nouvelles et des anciennes analyses dynamiques, l'algorithme appelle les plugins d'instrumentation en fonction des données d'exécution requises afin d'insérer ou de retirer les sondes de la plate-forme.

La seconde partie de l'outil d'instrumentation correspond à l'implémentation des plugins d'instrumentation. Chaque plugin doit implémenter l'interface générique `IPlatFormSensor`, représentée dans la figure 8.10. Cette interface est composée de deux méthodes. La méthode

insert ajoute une nouvelle sonde dans le système afin d'observer l'événement event passé en paramètre. La méthode remove retire la sonde du système qui observait l'événement event.

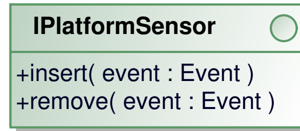

FIG. 8.10: Interface IPlatFormSensor

La version actuelle de CALICO contient quatre plugins d'instrumentation, dont deux pour la QdS :

Structure. L'observation des événements structurels correspond à tisser des aspects afin d'intercepter les appels à l'API de déploiement. Pour réaliser ce tissage, nous avons utilisé Spoon [Paw06] [9].

Trace. La capture de la trace repose sur un tissage d'aspect afin d'intercepter les messages entrants et sortants. Cependant, ce tissage a besoin de modifier la signature des opérations. Cette fonctionnalité n'est pas supportée par tous les tisseurs d'aspect. Nous avons donc utilisé Spoon.

QdS. Afin de capturer les événements de QdS, nous avons intégré dans CALICO le canevas de développement de sondes de QdS WildCAT [DL05] [10]. CALICO prend donc en charge l'observation de la charge CPU et du temps système.

Le tableau 8.4 donne le nombre de lignes de la partie commune et des plugins. D'un point de vue quantitatif, nous pouvons remarquer qu'un plugin est relativement simple et ne contient que peu de lignes. En effet, le rôle d'un plugin consiste à encapsuler un canevas de sondes existant. Concrètement, le plugin invoque l'API du canevas de sondes afin d'insérer une sonde dans le logiciel.

	Partie commune	structure	trace	QdS (WildCAT)	Total
Nb de lignes	498	103	376	184	1161

TAB. 8.4: Nombre de lignes de code pour l'outil d'instrumentation

D) Outil de chargement

Le rôle de l'outil de chargement est de déployer l'application sur une plate-forme cible. Cet outil est constitué de trois étapes : l'étape de comparaison des modèles de structure du système, l'étape d'ajustement et l'étape d'interprétation, comme cela a été expliqué dans la section 7.4.

L'implémentation de l'étape de comparaison des modèles est contenue dans le projet Eclipse CALICO_common. Pour rappel, cette étape compare les modèles de structure du système et produit le modèle de mise à jour qui spécifie la liste des opérations de construction, comme ajouter ou supprimer une entité.

L'implémentation de l'étape d'ajustement se base sur un mécanisme de plugin. L'implémentation d'un filtre consiste à développer une classe qui implémente l'interface IFilter représentée sur la figure 8.11. Cette interface contient une méthode transform qui prend en entrée le modèle de mise à jour et produit en sortie un nouveau modèle de mise à jour.

[9] http ://spoon.gforge.inria.fr
[10] http ://wildcat.ow2.org

L'étape d'interprétation est effectuée par un driver qui appelle l'API de déploiement de la plate-forme en fonction des opérations contenu dans le modèle de mise. Le driver est chargé dynamiquement en fonction de la plate-forme cible spécifiée par l'architecte dans le modèle d'intégration. Chaque driver doit implémenter l'interface générique `IPlatformDriver` représentée dans la figure 8.12. Cette interface est composée de quatre méthodes. Les méthodes `stop` et `start` correspondent, respectivement, à arrêter le système avant de débuter une adaptation dynamique et à redémarrer le système après l'adaptation. La méthode `process` prend en entrée une opération de construction du modèle de mise à jour. Elle a pour rôle d'appeler l'API de la plate-forme correspondant à cette opération. La méthode `init` initialise la plate-forme. Par exemple, dans le cas de CCM, elle se connecte à l'ORB de la plate-forme et récupère le service de nommage.

FIG. 8.11: Interface `IFilter`

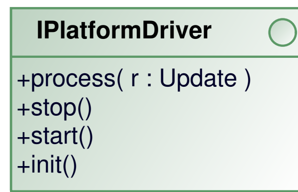

FIG. 8.12: Interface `IPlatformDriver`

L'implémentation courante de CALICO gère cinq plates-formes d'exécution : Fractal, OpenCOM, OpenCCM, FraSCAti et les services WEB. De plus, c'est le premier canevas capable de déployer incrémentalement un système SCA. D'un point de vue quantitatif, le tableau 8.3 résume le nombre de lignes de code de la partie commune, c'est-à-dire de l'étape de comparaison et de l'étape d'ajustement, et de chaque driver. La partie commune contient beaucoup plus de ligne de code que les drivers et est plus complexe. La complexité d'un driver dépend, quant à elle, de l'API de déploiement fourni par la plate-forme. Par exemple, FraSCAti n'expose pas d'API de déploiement de haut niveau, nous avons donc dû manipuler les concepts internes à la plate-forme.

E) Outil de débogage

Le rôle de l'outil de débogage est de recevoir des informations d'exécution provenant du système et de déclencher les analyses dynamiques appropriées. Afin de pouvoir prendre en compte les systèmes répartis, l'outil de débogage expose une interface Java RMI (RMI pour *"Remote Method Invocation"*) `ISensor`, représentée dans la figure 8.13. Cette interface contient une

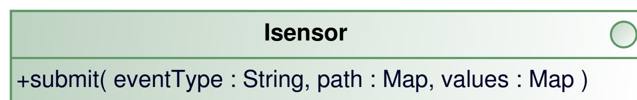

FIG. 8.13: Interface `ISensor`

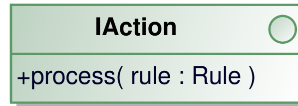

FIG. 8.14: Interface `IAction`

méthode `submit` qui permet au système d'envoyer des informations d'exécution. Le paramètre `evenType` spécifie le type d'événement. Le paramètre `path` est une table de hachage de chaîne de caractères qui identifie les éléments systèmes qui ont émis l'événement. Le paramètre `values` est une table de hachage contenant la valeur des données d'exécution. Afin de prendre en compte des plates-formes écrites dans un langage de programmation autre que Java, une surcouche WEB service pourra être ajoutée au dessus de l'interface RMI.

Chaque fois que l'outil de débogage reçoit des informations d'exécution, il identifie l'analyse dynamique correspondante à effectuer. Cette analyse dynamique teste sa condition. Lorsque la condition est vraie, l'analyse dynamique déclenche l'action de finalisation de la validation. Afin d'être extensible, nous avons défini une interface générique `IAction`, représentée dans la figure 8.14, qui doit être implémentée par chaque action. Cette interface possède une méthode `process` qui prend en entrée l'analyse dynamique qui a déclenché l'action à exécuter.

L'implémentation de l'outil de débogage est isolée dans le projet Eclipse `CALICO_adaptationEngine`. D'un point de vue quantitatif, elle correspond à 654 lignes de code Java.

8.1.3 Bilan

L'implémentation actuelle de CALICO est disponible librement sur la forge INRIA [11] sous licence LGPL [12]. Elle supporte cinq plates-formes d'exécution : Fractal, OpenCCM, OpenCOM, FraSCAti et services WEB. Elle prend en charge les quatre catégories d'analyse, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. De plus, elle est capable de capturer la trace des messages échangés entre les composants, ainsi que la charge CPU et le temps système grâce à l'intégration du canevas WildCAT.

CALICO a été conçu afin d'être extensible, c'est-à-dire autoriser l'intégration d'un maximum de plates-formes, d'outils d'analyse et de canevas de sondes existants. D'une part, nous avons défini un ensemble d'API génériques documentées pour permettre à des développeurs d'intégrer de nouveaux plugins d'analyse, de génération de code, d'instrumentation, de déploiement, etc. D'autre part, nous avons mis en œuvre un mécanisme de chargement dynamique de ces plugins afin de prendre en compte les nouvelles extensions sans avoir besoin de redémarrer CALICO. De plus, la majorité du code est factorisée pour les différentes plates-formes. Ainsi, le développement d'extension de CALICO est simplifié et ne nécessite que peu de lignes de code.

D'un point de vue quantitatif, l'implémentation de CALICO est répartie dans 16 projets Eclipse pour un total d'environ 13000 lignes de code Java et 8000 de XML, à partir desquelles 90000 lignes de code Java sont générées en plus. En outre, afin d'augmenter la visibilité de CALICO, nous avons mis en place un site Web contenant de la documentation, des tutoriels et des captures d'écran expliquant comment utiliser CALICO.

8.2 Évaluation de CALICO

Cette section présente les évaluations que nous avons faites avec l'implémentation de CALICO. Dans une première section, nous décrivons les expérimentations que nous avons effectuées afin de valider l'extensibilité de CALICO. Dans une seconde section, nous donnons

[11] http ://calico.gforge.inria.fr
[12] http ://www.gnu.org/licenses/old-licenses/gpl-2.0.html

les évaluations de performances de CALICO pour la validation dynamique de système. Dans une troisième partie, nous discutons des gains qu'apporte CALICO aux différents acteurs du développement du logiciel.

8.2.1 Extensibilité

Cette section présente nos évaluations sur l'extensibilité de CALICO. Nous avons mesuré le niveau de complexité pour ajouter une nouvelle plate-forme dans CALICO. La première partie de cette section résume les différentes étapes requises pour étendre CALICO. La seconde partie valide les API génériques de CALICO. Elle vise à étudier si les API génériques sont suffisantes pour ajouter différentes plates-formes et différents outils existants. La troisième partie évalue la quantité de travail nécessaire pour ajouter le support d'une nouvelle plate-forme.

A) Aperçu

La figure 8.15 présente les trois étapes à effectuer pour ajouter le support d'une nouvelle plate-forme dans CALICO. La première étape correspond à l'écriture du profil de la plate-forme, c'est-à-dire le fichier OCL exprimant toutes les contraintes de la plate-forme que toute architecture doit satisfaire. Le profil est automatiquement pris en compte durant l'analyse des interactions afin de valider si l'architecture décrite respecte toutes les règles de composition de la plate-forme. La seconde étape concerne la génération de code. Elle consiste à implémenter le plugin de génération de code dédié à la plate-forme. Concrètement, cette implémentation correspond à écrire une classe qui étend la classe abstraite `Model2CodeGeneration`. Enfin la troisième étape consiste à développer le driver chargé de déployer le système sur la plate-forme. Pour réaliser cette étape, il faut créer une classe qui implémente l'interface `IPlatformDriver` dont le rôle est d'interpréter les opérations de construction du modèle de mise à jour en appelant l'API de déploiement de la plate-forme. Si nécessaire, le développeur peut implémenter de nouveaux filtres d'ajustement permettant de modifier le modèle de mise à jour afin qu'il satisfasse les contraintes de déploiement de la plate-forme.

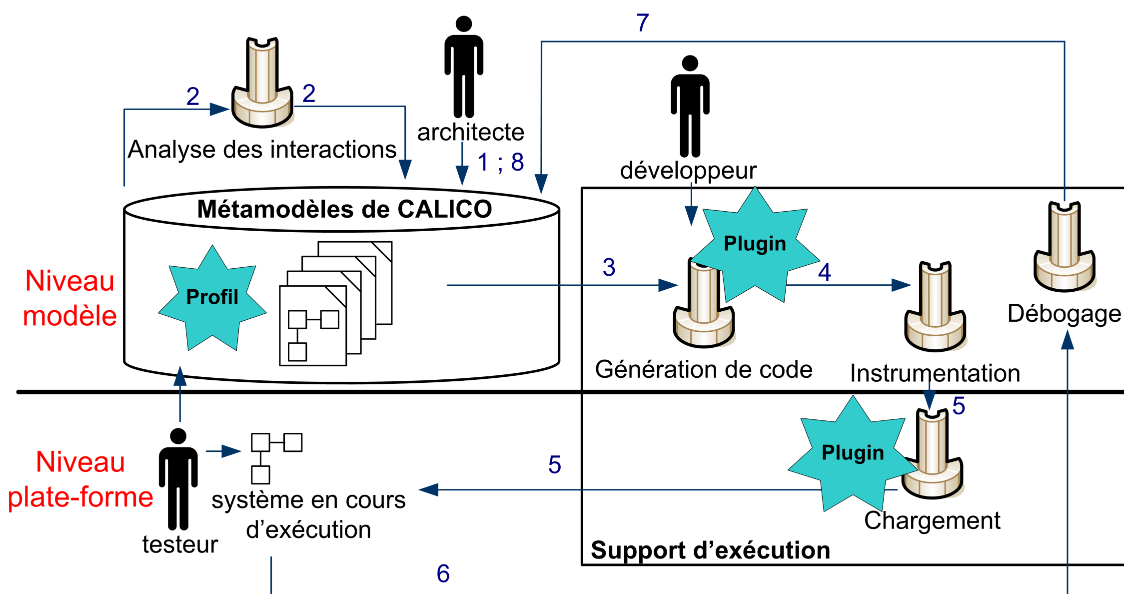

FIG. 8.15: Extension de CALICO

B) APIs génériques

Dans un premier temps, nous avons effectué des expérimentations afin de valider les APIs génériques de CALICO. L'objectif est de valider les APIs concernant l'ajout de nouvelles analyses, de nouvelles sondes et de nouvelles plates-formes. Pour valider l'API d'ajout de nouvelles analyses statiques et dynamiques, nous avons intégré dans CALICO différents outils d'analyse englobant les quatre catégories d'analyse, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. De plus, nous avons choisi d'intégrer des outils qui reposent sur des technologies différentes, comme EMF-OCL, Fractal-BPC, Maxima, etc. Ensuite, pour valider l'API correspondant à l'ajout de nouvelles sondes, nous avons intégré dans CALICO différents types de sondes qui utilisent des technologies différentes, comme par exemple le tissage d'aspect, la génération dynamique de code ou l'encapsulation de canevas de sondes différentes. Enfin, pour valider l'API concernant l'ajout de nouvelles plates-formes, nous avons ajouté le support de cinq plates-formes. Nous avons choisi d'utiliser des plates-formes très différentes. C'est pourquoi, nous avons intégré des plates-formes académiques, comme Fractal ou OpenCOM, et des plates-formes industrielles, comme OpenCCM et FraSCAti. De plus, nous avons pris en compte, à la fois les plates-formes à composants et les plates-formes orientées services.

Bilan. Chacune des intégrations a parfaitement réussi. Nos expérimentations n'ont pas mis en évidence de point bloquant. Nous avons exploité le résultat des intégrations pour améliorer et ajuster les APIs génériques de CALICO. Nous avons ainsi pu valider les APIs génériques en appliquant une démarche pragmatique.

C) Quantité de travail

Dans un second temps, nous avons évalué la quantité de travail requise pour ajouter le support d'une nouvelle plate-forme. Comme il est très difficile d'effectuer des mesures sur le niveau de complexité d'une implémentation, nous avons effectué deux expérimentations différentes afin de prendre en compte l'expérience du développeur. D'abord, nous avons mesuré le temps de travail qui est requis par un spécialiste de CALICO pour ajouter une nouvelle pate-forme. Ensuite, nous avons évalué la faisabilité pour un développeur débutant d'ajouter une nouvelle plate-forme.

Le développeur spécialiste. Nous avons mesuré le temps requis par un spécialiste de la plate-forme CALICO pour ajouter le support de la plate-forme FraSCAti. Cet expert n'avait à priori aucune connaissance de la plate-forme FraSCAti. Son travail a donc d'abord consisté à prendre connaissance du modèle SCA et de la plate-forme FracSCAti. Pour commencer, il a étudié la spécification de SCA afin de définir le profil de la plate-forme. Ensuite, il a examiné les exemples de FraSCAti afin d'écrire le plugin de génération de code. Pour finir, il s'est entretenu avec le développeur de la plate-forme et il a étudié le code source de la plate-forme afin de pouvoir implémenter le driver. Au total, le spécialiste a mis cinq jours pour intégrer la plate-forme FraSCAti. Nous considérons que ce temps correspond au temps minimum pour ajouter une nouvelle plate-forme et que donc le temps requis pour ajouter une plate-forme est supérieur à une semaine.

Le développeur débutant. Nous avons évalué la quantité de travail qui est requise par un développeur débutant pour ajouter le support d'une nouvelle plate-forme. Concrètement, nous avons proposé un projet technique du second semestre de Master 1 à un étudiant dont l'objectif est d'ajouter le support des services WEB [Wit09]. L'intégration du support des services WEB n'est pas triviale dans la mesure où la très grande partie des plates-formes d'exécution des services WEB offre généralement moins de fonctionnalités de reconfigurations que les plates-formes

à composants. Par exemple, les plates-formes d'exécution industrielles comme Axis [13] et GlassFish [14] ne prennent en charge que le déploiement et le repliement des services WEB.

Le projet technique s'est déroulé en deux étapes. Dans un premier temps, puisque l'étudiant ne possédait aucune connaissance sur les services WEB, ni sur les outils de CALICO, il a fait une étude des services WEB et il a comparé deux plates-formes d'exécution différentes : Axis et GlassFish. Il s'est focalisé sur les implémentations en Java des services WEB et il a donc examiné le standard JSR 181 [SUN06a]. Il a donc décidé d'intégrer GlassFish dans CALICO car c'est la plate-forme de référence de SUN. Dans un second temps, à partir des connaissances acquises, il a défini le profil OCL pour les services WEB. Puis, il a implémenté le plugin de génération de code, le filtre d'ajustement `WEBServiceOrder` et le driver dédié aux services WEB. Pour réaliser ce travail, il a eu besoin d'exploiter la technologie de tissage d'aspect, qu'il ne connaissait pas. Il a donc appris à manipuler Spoon.

Bilan. Ces deux petites expériences n'ont pas mis en évidence de point bloquant. Le test d'intégration des services WEB a été très pertinent dans la mesure où l'architecture des services WEB est très différente des architectures à composants. Nous en avons déduit que la complexité d'ajout d'une nouvelle plate-forme dépend très fortement des fonctionnalités de déploiement offertes par la plate-forme. La tâche la plus complexe consiste à comprendre le fonctionnement de la plate-forme. La prise en main du canevas CALICO est rapide. En effet, le développeur n'a pas besoin de modifier une seule ligne de code existant, les APIs génériques exportées par CALICO sont suffisantes pour l'ajout d'une plate-forme. De plus, le support de la plate-forme est clairement isolé dans un plugin Eclipse distinct. Mais aussi, dans les cas où les plates-formes ont des similarités, le développeur peut s'inspirer des drivers existants pour élaborer le nouveau driver.

8.2.2 Performance

Nous avons évalué les performances de chacun des outils de CALICO. Tous les tests ont été effectués sur un portable doté d'un processeur Intel®Core™2 Duo cadencé à 1,33GHz. La machine virtuelle Java de SUN®version 1.6.0_11 a été utilisée.

Les expérimentations ont été effectuées sur la plate-forme Fractal. La structure du système qui a servi aux expérimentations est représentée dans la figure 8.16. C'est une architecture hiérarchique de forme arborescente. Le plus haut niveau hiérarchique de l'architecture est un composant composite. Il contient un composant primitif connecté avec deux autres composants composite `A`, comme le montre la figure 8.16. Récursivement, la structure interne du composite `A` est la même que celle du composite parent. Durant les expérimentations, pour faire varier la taille du système de test, nous avons modifié le nombre de niveaux hiérarchiques.

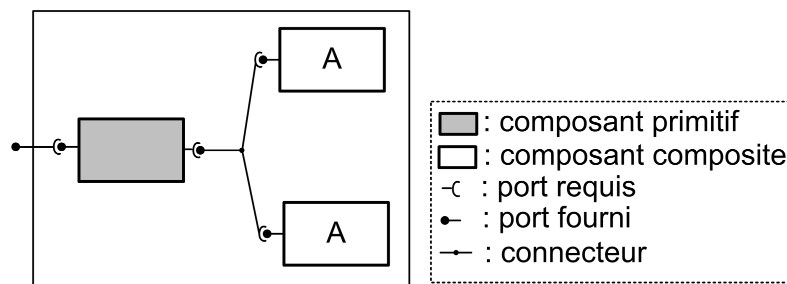

FIG. 8.16: Architecture du système de test

[13] http ://ws.apache.org/axis/
[14] https ://glassfish.dev.java.net

A) Analyse du système

Nous avons mesuré le temps mis par l'outil d'analyse des interactions qui vérifie que les propriétés applicatives spécifiées par l'architecte sont respectées. La figure 8.17 représente le temps mis en secondes pour analyser les propriétés structurelles et comportementales du système en fonction du nombre de composants. L'analyse structurelle est très rapide, elle dure moins de 2,5s pour analyser un système de 1500 composants. Son coût est donc négligeable. L'analyse du comportement est, quant à elle, beaucoup plus longue. Elle dure environ 10s pour un système de 1500 composants. Néanmoins, cela n'est pas surprenant, les analyses comportementales sont beaucoup plus complexes. Donc, afin de ne pas contraindre l'architecte à attendre longtemps le résultat de la validation dynamique de son système, nous lui permettons de sélectionner les analyses qu'il désire appliquer sur son système.

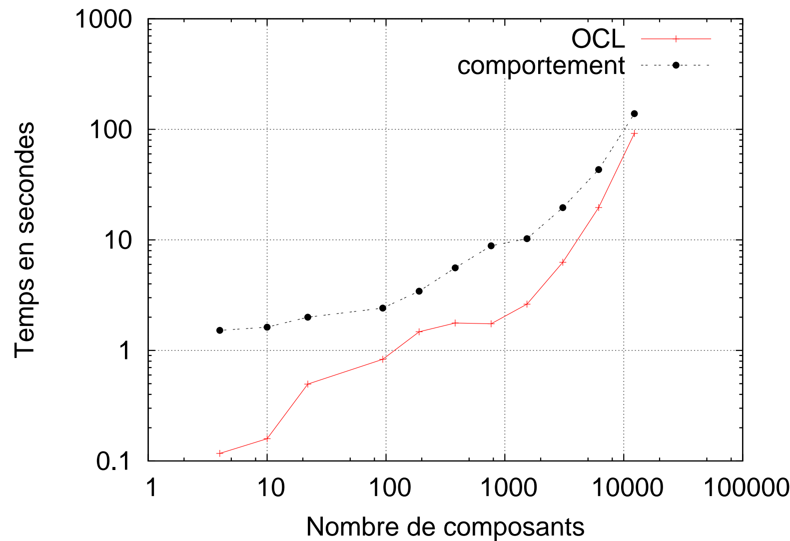

FIG. 8.17: Durée d'analyse du système

B) Déploiement du système

Ensuite, nous avons évalué le surcoût de CALICO pour déployer le système. Ainsi, nous avons comparé le temps mis pour déployer le système avec l'outil de chargement de CALICO, par rapport au temps mis avec l'outil de déploiement natif Fractal ADL. La figure 8.18 montre le temps mis en secondes pour déployer le système en fonction du nombre de composants. Dans le cas du test avec Fractal ADL, la mesure du temps de déploiement prend en compte le temps mis pour lire le fichier fractal ADL et instancier les composants sur la plate-forme. Dans le cas du test avec l'outil de chargement de CALICO, la mesure du temps prend en compte la lecture du fichier de description EMF et l'instanciation des composants sur la plate-forme. Les tests mettent en évidence que CALICO est aussi performant que l'outil natif pour déployer un système. Par exemple, pour déployer un système de 3000 composants, CALICO met 14 secondes alors que Fractal ADL met 23 secondes.

Ensuite, nous avons mesuré le temps mis pour effectuer un déploiement incrémental. Pour faire cette évaluation, nous partons d'un système déjà déployé dans lequel nous ajoutons un nouveau composant primitif dans le composite racine du système. Afin d'évaluer l'impact de la taille du système sur le temps, nous faisons évoluer la taille du système déjà déployé, comme le montre la figure 8.19. Jusqu'à 3000 composants, la complexité est linéaire et le temps obtenu est raisonnable pour une activité de validation dynamique du système. CALICO met environ 6

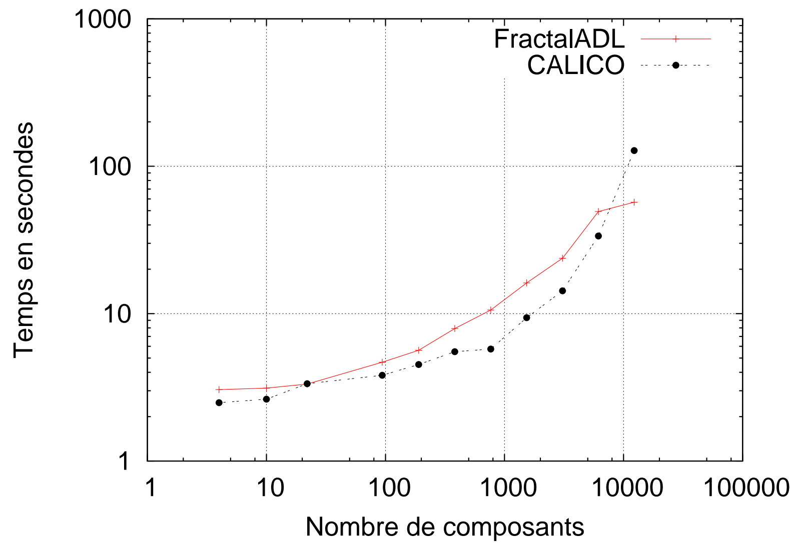

FIG. 8.18: Durée du déploiement de tous le système

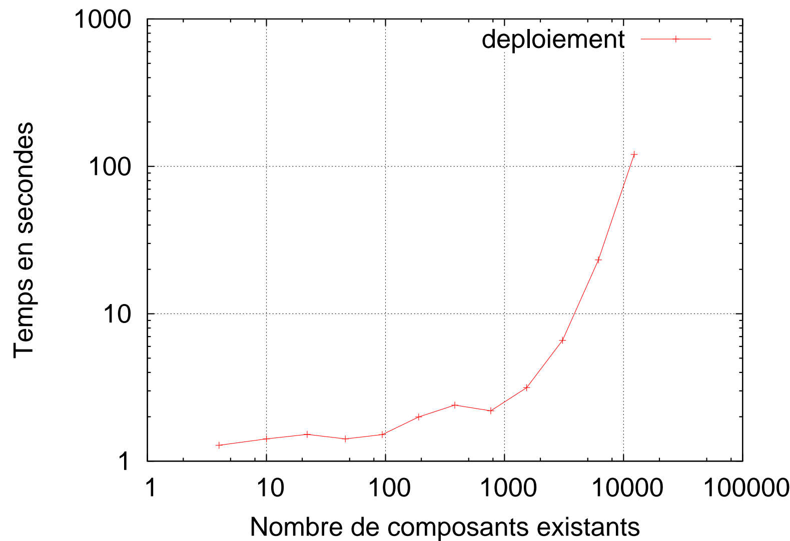

FIG. 8.19: Temps mis pour déployer un nouveau composant dans le système

secondes pour comparer les deux modèles de structure du système, générer le modèle de mise à jour, exécuter tous les filtres d'adaptation et pour interpréter le modèle de mise à jour.

C) Exécution du système

Durant l'exécution du système, nous avons aussi évalué la consommation mémoire du canevas CALICO et du système. La figure 8.20 représente la consommation mémoire en mégaoctet en fonction du nombre de composants du système. Sur la figure, la réduction de la consommation mémoire correspond à la libération de la mémoire par le ramasse miette de Java. L'augmentation de la mémoire utilisée est linéaire. Pour 3000 composants, la consommation mémoire de CALICO et du système est de 47 mégaoctets.

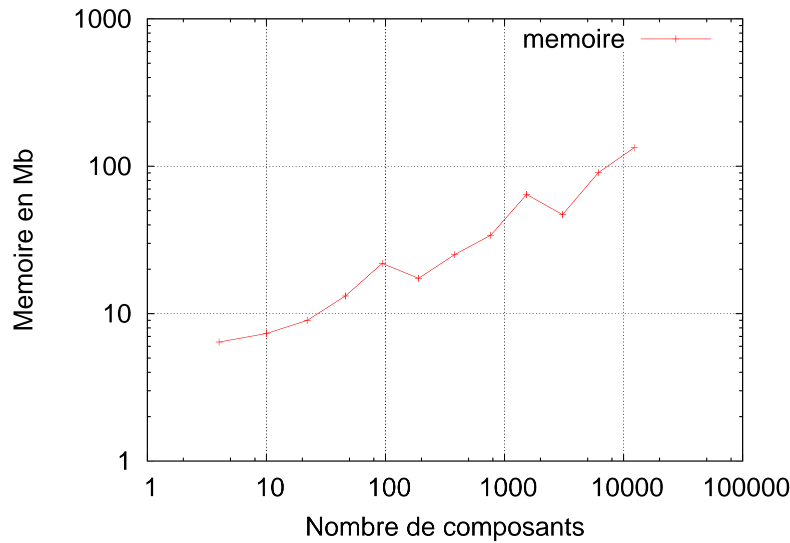

FIG. 8.20: Consommation mémoire de CALICO

D) Réification des événements

Nous avons mesuré le temps mis pour envoyer les événements du niveau plate-forme au niveau modèle lors de l'étape de validation dynamique. Pour réaliser les mesures, nous avons conçu une architecture linéaire de type *"pipe and filter"* qui est représentée sur la figure 8.21. Elle est composée d'un composant client qui envoie une requête, d'un composant serveur qui effectue une addition de deux entiers, et de 500 composants filtres F_i situés entre le client et le serveur. Nous avons mesuré le temps mis pour réifier les messages qui entrent et sortent de chaque filtre. Afin de permettre l'observation des événements, CALICO ajoute automatiquement un composant sonde S_i avant et après chaque port des composants filtre, comme le montre la figure 8.21.

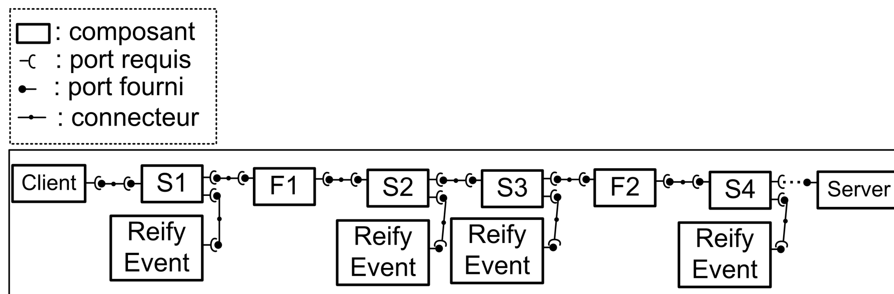

FIG. 8.21: Architecture du système *"pipe and filter"*

CALICO propose à l'architecte de choisir entre deux modes de génération de sonde : le mode externe et le mode interne. Avec le mode externe, la sonde envoie chaque message au niveau modèle, sous forme d'événement. Le message est alors analysé au niveau modèle afin de détecter s'il déclenche l'exécution d'une analyse dynamique. Avec le mode interne, la condition associée à l'analyse dynamique est directement générée dans la sonde. De cette manière, la sonde envoie le message au niveau modèle uniquement si la condition est vraie afin que l'analyse dynamique finalise la validation.

La figure 8.22 représente la durée en millisecondes que met le composant client à invoquer l'opération d'addition du composant serveur, en fonction du nombre de sondes dans le système.

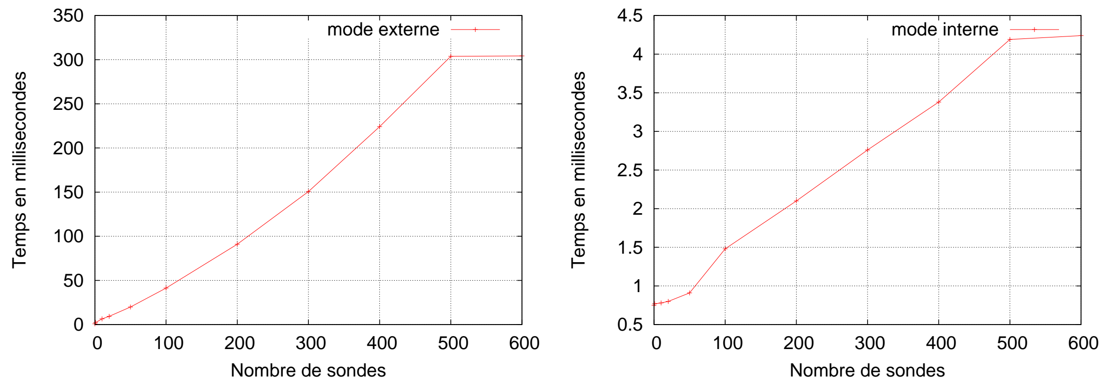

FIG. 8.22: Durée de la réification des événements

La partie gauche de la figure correspond au mode externe et la partie droite au mode interne. Dans ce scénario, le pire des cas est atteint avec 600 sondes car cela provoque un dépassement de la pile d'appel des méthodes de la machine virtuelle Java.

Le mode de génération interne n'a presque aucun impact sur le temps mis par l'appel de méthode. Le temps de l'appel est de 0,75 millisecondes sans sonde et de 4 millisecondes dans le pire des cas avec 600 sondes. Cependant avec ce mode, les sondes doivent être repliées, régénérées, recompilées et redéployées chaque fois que l'architecte modifie la condition associée aux analyses dynamiques. En conséquence ce mode est adéquat lorsque l'expression de la condition associée à l'analyse est stable.

Le mode de génération externe a un impact plus important. Ce coût est dû à la communication distante RMI entre le niveau modèle et le niveau plate-forme qui est effectuée par chaque sonde. Par exemple, dans le pire des cas, la durée de l'appel de la méthode prend 304 millisecondes (cf. figure 8.22). Donc, avec ce mode le système est plus lent à s'exécuter. Cependant, lorsque l'architecte ajuste l'expression de la condition d'une analyse dynamique, le changement prend directement effet dans le système. En conséquence, ce mode est adapté pour la phase de conception et de mise au point du système, comme par exemple, lorsque l'architecte ajuste le profil des terminaux des professionnels de santé dans le système DMP. De plus, dans la pratique, le nombre maximum de sondes dans un flot de données ne dépasse que très rarement la dizaine. Dans ce cas là, le temps pris par l'appel dure moins de 6 millisecondes. L'approche est donc toujours utilisable pour valider dynamiquement le système.

Le coût mis par la réification des événements de QdS est directement dépendant de l'efficacité du canevas de sondes qui est utilisé [CELQ04]. CALICO introduit un unique surcoût qui est lié à la communication distante entre le niveau plate-forme et le niveau modèle. Cependant la communication distante est indispensable afin de pouvoir prendre en charge les systèmes répartis.

E) Bilan

Nous avons choisi d'effectuer des tests de performance qui pousse CALICO jusque dans ses limites. Les mesures démontrent que CALICO est utilisable pour valider dynamiquement des systèmes. CALICO offre des performances correctes jusqu'à 10000 composants, ce qui correspond à des architectures de grandes tailles. D'ailleurs, toutes les plates-formes d'exécution ne sont pas capables de prendre en charge des architectures aussi grandes. Par exemple, la version 1.3.2 de la plate-forme d'exécution de référence de SCA, Tuscany [15], qui est développée par Apache, ne peut gérer que jusqu'à 6000 composants. En effet, dans la pratique, la plus grande majorité des architectures à composants contiennent largement moins de 10000 composants.

Pour les très larges systèmes, c'est-à-dire les systèmes qui contiennent plus de 15000 composants, la durée mise par CALICO pour valider dynamiquement le système devient exponentielle.

[15] http ://tuscany.apache.org

Afin de déterminer l'origine du problème, nous avons fait varier la quantité de mémoire maximale de la machine virtuelle Java (JVM pour "*Java Virtual Machine*"). Nous avons constaté que si nous augmentons la quantité mémoire, la durée du déploiement du système diminue. Par exemple, en doublant la quantité mémoire de la JVM, la durée du déploiement de 1500 composants diminue de 30 secondes. Nous supposons donc que l'origine du problème provient du canevas EMF qui met les éléments du modèle en cache lorsque la mémoire devient rare. En conséquence, le temps requis pour manipuler le modèle augmente très fortement à cause des accès au disque dur.

8.2.3 Apport aux acteurs du développement

CALICO offre aux différents acteurs du développement du logiciel un cadre pour faire évoluer de manière fiable le logiciel. Il les guide dans leur développement en les faisant suivre un cycle de développement itératif et incrémental et en changeant le moins possible leurs habitudes. L'utilisation de CALICO est la plus transparente possible.

Architecte. Pour concevoir une architecture logicielle fiable, l'architecte a besoin de spécifier les diverses exigences que l'application doit satisfaire. L'architecture doit ensuite être analysée afin de détecter si les exigences sont violées. Pour réaliser cet objectif, CALICO apporte à l'architecte une palette de moyen de spécification lui permettant de décrire les quatre catégories de propriétés applicatives de son logiciel, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. A notre connaissance, en dehors de CALICO, aucune plate-forme à composants et orientée services ne le permet. Ainsi, grâce à CALICO, l'architecte peut exploiter des outils de spécification et d'analyse sur des plates-formes qui n'en offrent pas de base, ce qui lui permet de concevoir une architecture plus fiable. De plus, CALICO garde la description de l'architecture synchronisée avec le logiciel en cours d'exécution, ce qui permet à l'architecte de raisonner sur une architecture qui reflète l'état du logiciel. Cette contribution est l'élément clé qui permet à l'architecte de faire facilement évoluer son architecture.

Afin d'être le plus transparent possible, CALICO autorise l'architecte à exploiter le formalisme de description qu'il a l'habitude de se servir pour concevoir une architecture logicielle. Il peut donc réutiliser les différentes vues graphiques ou textuelles qu'il connaît.

Développeur. Grâce à CALICO, le développeur n'a besoin que d'écrire le code métier des composants et des services. Le code technique est entièrement généré par CALICO. Donc contrairement à l'approche de programmation par attributs, le développeur a la garantie que le code technique est cohérent avec la description de l'architecture faite par l'architecte. En outre, CALICO soulage le développeur de l'intégration, dans le code des composants et des services, des mécanismes requis pour valider dynamiquement le logiciel.

Pour ne pas être intrusif, CALICO laisse le développeur choisir l'outil de développement et l'outil de compilation qu'il connaît. De plus, dans la mesure où CALICO est multi plates-formes, il n'impose pas non plus l'utilisation d'une plate-forme d'exécution.

Administrateur système. Grâce à CALICO, l'administrateur n'a plus besoin de déployer les composants applicatifs. Néanmoins, il a toujours la tâche de mettre en place les plates-formes d'exécution sur les différentes machines.

Testeur. CALICO offre un cadre aux testeurs pour valider dynamiquement le logiciel. Concrètement, les testeurs peuvent définir de nouvelles analyses dynamiques à exécuter sans qu'ils aient besoin de modifier manuellement le code du logiciel. En effet, pour chaque nouvelle analyse dynamique ajoutée dans le modèle de débogage, CALICO met automatiquement en œuvre le mécanisme requis dans le logiciel pour l'exécution de cette analyse.

8.3 Conclusion et perspectives

Afin de concrétiser nos travaux sur CALICO, nous avons élaboré une implémentation conséquente de CALICO. Nous avons donc produit un canevas de développement itératif et incrémental pour concevoir et faire évoluer les architectures à composants et les architectures orientées services. Notre implémentation de CALICO est complètement intégrée dans l'IDE Eclipse. Elle prend en charge cinq plates-formes et est capable d'effectuer les quatre catégories d'analyse, c'est-à-dire structurelles, comportementales, de flot de données et de QdS. Elle offre aussi des vues graphiques permettant à un architecte de concevoir son système.

Un travail important a été effectué pour permettre l'extensibilité de CALICO. Nous avons donc mis au point et validé les APIs génériques de CALICO en intégrant quatre plates-formes à composants, une plate-forme orientée services, quatre outils d'analyse et trois plugins d'instrumentation. De plus, nous avons porté une attention toute particulière pour réduire au maximum la quantité de code à écrire pour étendre CALICO. Concrètement, nous avons factorisé tout le code commun à différentes plates-formes dans les bibliothèques de CALICO. Par exemple, le support de la plate-forme Fractal est constitué de 20% de code spécifique et 80% de code générique. D'autre part, CALICO est un canevas adaptable, dans le sens où il offre la possibilité d'ajouter le support de nouvelles plates-formes, analyses, sondes, etc. pendant son exécution et donc sans interrompre l'architecte logiciel dans son travail de conception et de validation dynamique.

Les premières évaluations expérimentales de CALICO ont mesuré les temps d'analyse, de déploiement et de réification des événements d'un système, ainsi que la consommation mémoire de CALICO. Ces expérimentations ont pour objectif de tester le passage à l'échelle de CALICO. Elles démontrent que CALICO offre des performances satisfaisantes pour valider dynamiquement des grands systèmes jusqu'à 10000 composants.

Globalement, CALICO est un canevas intégrateur, capable de supporter de multiples technologies différentes. Par exemple, CALICO regroupe, au sein d'un même canevas de développement, les différentes plates-formes à composants qui ont été développées dans l'équipe Jacquard/ADAM, c'est-à-dire OpenCCM, Fractal et FraSCAti. De plus, grâce au mécanisme de co-évolution entre le niveau modèle, décrivant le système, et le niveau plate-forme qui contient le système en cours d'exécution, CALICO offre à l'architecte une vue de haut niveau d'abstraction qui est toujours synchronisée avec le système. Ce couplage entre une vue abstraite et le système réel permet de rapprocher le monde des analyses et le monde des intergicielles.

En perspective, nous désirons que CALICO devienne une plate-forme d'expérimentation. CALICO a le potentiel de fédérer plusieurs travaux de l'équipe. Par exemple, grâce sa flexibilité, CALICO peut permettre le prototypage rapide de démonstrateurs scientifiques et peut aussi faciliter la diffusion de résultats scientifiques. Nous avons notamment obtenu, à la fin de la thèse, un financement ADT INRIA (ADT pour *"Action de Développement Technologique"*) pour qu'un ingénieur travail sur l'implémentation de CALICO. Sa mission sera, dans un premier temps, d'effectuer une maintenance corrective du code actuel de CALICO, librement disponible sur la forge INRIA, afin d'obtenir un noyau plus stable. Cette tâche est indispensable pour atteindre notre objectif de canevas fédérateur. Dans un second temps, l'ingénieur aura pour objectif de favoriser la diffusion de CALICO à l'extérieur. Il pourra pour cela enrichir la documentation et les tutoriels existants. Il aura aussi la possibilité d'effectuer des démonstrations lors de conférences internationales.

Chapitre 9

Conclusion et perspectives

Sommaire

9.1 Résumé des contributions 175

9.2 Perspectives 177

9.2.1 Perspectives à court terme 177

9.2.2 Axes de recherches 178

Les systèmes logiciels modernes se distinguent par un besoin d'évolutions rapides. Ces évolutions sont nécessaires pour prendre en compte les nouvelles exigences des utilisateurs. De plus, un logiciel a besoin de constamment satisfaire les contraintes de fiabilité et de qualité de services exigées. Face à ces enjeux, les architectures logicielles et la méthode de développement agile facilitent l'évolution rapide de logiciels complexes. Néanmoins, en dépit de toutes ces approches, la fiabilisation des évolutions reste un défi.

Nous avons présenté dans ce mémoire un canevas de développement agile dont l'objectif est de permettre l'évolution fiable de logiciels à composants ou orientés services. La section 9.1 résume nos contributions et la section 9.2 donne quelques perspectives de recherches.

9.1 Résumé des contributions

Nous avons adressé la problématique soulevée dans l'introduction en proposant un canevas de développement agile pour la conception et l'évolution fiable d'architectures logicielles. Notre proposition remplit les trois points énoncés dans l'introduction, à savoir (1) permettre l'évolution d'architectures à composants et orientées services, (2) fiabiliser leurs évolutions et (3) surmonter le manque de fonctionnalité de validation des plates-formes d'exécution. Nous avons aussi concrétisé et évalué notre proposition. Les paragraphes suivants détaillent les principales contributions de cette thèse.

A) Evolution d'architectures à composants et orientées services

CALICO permet aux différents acteurs du développement du logiciel de faire évoluer leur logiciel en ajoutant de nouvelles fonctionnalités et en supprimant les fonctionnalités obsolètes. Ces évolutions peuvent avoir lieu aussi bien au moment de la conception que de l'exécution. Ce support repose, à la fois, sur la synchronisation de la vue conceptuelle de l'architecture avec le logiciel en cours d'exécution, et sur l'utilisation d'un cycle de développement itératif et incrémental. Ainsi, l'architecte dispose d'une vue conceptuelle qui reflète l'état du logiciel en cours d'exécution, ce qui lui permet de raisonner et de mettre au point plus rapidement son

logiciel. De cette manière, les acteurs développent leur système en itérant entre les étapes de conception, d'implémentation et de validation dynamique.

Concrètement, lors de la conception et de l'évolution de son système, l'architecte dispose d'un ensemble de métamodèles pour décrire la structure de l'architecture et ses diverses exigences applicatives et de qualité de services. Ces métamodèles ont été établis à partir d'une étude de domaine sur 16 modèles à composants et à services dans le but d'offrir une solution générique et indépendante de toute plate-forme. Lors de l'implémentation, CALICO génère le squelette de code des composants/services de telle sorte que les développeurs ont seulement besoin de coder le métier des composants et des services. Afin de permettre la validation dynamique du logiciel par les testeurs, CALICO met automatiquement en oeuvre les analyses dynamiques du logiciel en instrumentant le code de l'application, avant de déployer le système sur la plate-forme d'exécution cible. Ainsi, les testeurs peuvent se concentrer sur l'exécution des scénarios d'utilisation du logiciel.

B) Fiabilisation des évolutions

Pour concevoir un logiciel fiable, nous permettons à l'architecte de spécifier les exigences de son application. Pour que l'architecte puisse décrire ses exigences le plus précisément possible, CALICO lui offre une palette de spécification qui couvre les quatre catégories de propriétés applicatives, c'est-à-dire structurelle, comportementale, de flot de données et de QdS. De cette manière, lors de chaque évolution, CALICO valide la compatibilité des interactions entre les composants et les services, et vérifie si les exigences applicatives sont toujours respectées. Concrètement, nous réalisons cet objectif, non pas en proposant de nouvelles analyses, mais en offrant un cadre fédérateur qui autorise la réutilisation de la plupart des outils d'analyse statique pour les architectures logicielles et des outils d'analyse dynamique du logiciel qui sont dispersés dans les différentes plates-formes existantes. Ce support repose sur une étude de domaine concernant les quatre catégories d'analyse. Nous en avons déduit les informations qui sont requises par ces analyses et nous avons élaboré notre canevas en accord avec ces besoins. Ainsi, lorsque l'architecte fait évoluer son architecture, les modifications ne sont propagées dans la plate-forme cible que si l'analyse statique ne détecte aucune violation des exigences. De plus, CALICO offre un cadre qui autorise l'intégration d'analyse incrémentale des évolutions, comme par exemple SFSP, afin que seules les évolutions effectuées par l'architecte soient analysées.

En outre, le mécanisme de validation que nous avons adopté est plus précis que la majorité des outils d'analyse existants qui ne prennent pas en compte les interactions partiellement compatibles. En effet, notre approche exploite la complémentarité des outils d'analyse statique et d'analyse dynamique. De cette manière, lorsque les analyses statiques ont besoin de données d'exécution pour finaliser la validation de l'architecture, comme par exemple connaître la valeur d'un message échangé, CALICO définit automatiquement les analyses dynamiques qui devront être exécutées pendant la validation dynamique du logiciel et met en oeuvre les mécanismes d'observation associés nécessaires. Notre solution réduit donc l'impact de performance lié à l'observation de données d'exécution car seules les données requises par les analyses dynamiques sont capturées.

Enfin, CALICO permet aussi de limiter l'introduction d'incohérences entre chaque étape du cycle de développement grâce au couplage très fort entre chaque étape. Cette solution facilite le développement en automatisant le maximum des tâches complexes qui sont sources d'erreurs, comme par exemple la génération du code technique conforme aux spécifications ou le déploiement des évolutions sur la plate-forme.

C) Surmonter le manque de fonctionnalité de validation des plates-formes d'exécution

Notre approche est indépendante de toute plate-forme d'exécution. En effet, CALICO permet de résoudre le problème lié au manque de fonctionnalité de validation des évolutions dans les plates-formes d'exécution. Notre solution est composée de trois parties. (1) CALICO externalise des plates-formes d'exécution la prise en compte des exigences, c'est-à-dire le

moyen de les spécifier et de les analyser statiquement. Ces exigences sont ainsi sauvegardées, indépendamment de toute plate-forme, dans les modèles qui sont gardés synchronisés avec le logiciel en exécution. (2) Les analyses dynamiques sont externalisées des plates-formes et ne reposent donc pas sur les fonctionnalités offertes par les plates-formes. (3) CALICO instrumente automatiquement le code de l'application pour permettre la capture des données d'exécution requises par les analyses dynamiques. L'ensemble de ces trois contributions permet d'offrir un support pour la validation statique et dynamique des évolutions à des plates-formes qui n'en disposent pas nativement.

D) Concrétisation, évaluation et expérimentation

Nos contributions se concrétisent par une implémentation multi plates-formes. La version actuelle prend en charge quatre plates-formes à composants et une plate-forme orientée services. Elle intègre différents travaux existants, comme par exemple les outils d'analyse OCL, les protocoles comportementaux de la plate-forme SOFA et le canevas de sonde WildCAT. En outre, notre implémentation a été conçue afin de pouvoir évoluer facilement, c'est-à-dire que notre implémentation supporte l'ajout de nouvelles extensions sans interrompre l'exécution de CALICO. De plus, les tests de performances que nous avons réalisés sur notre implémentation mettent en évidence le fait que CALICO offre des performances suffisantes pour faire évoluer des applications jusqu'à 10000 composants et services, ce qui correspond à la montée en charge maximale de nombreuses plates-formes existantes.

Afin de pérenniser notre implémentation, à la fin de la thèse, nous avons obtenu un financement ADT pour un ingénieur. Sa mission est de consolider l'implémentation de CALICO et d'y intégrer différents travaux de l'équipe afin que CALICO devienne un cadre fédérateur pour l'équipe-projet ADAM et qu'il permette le prototypage rapide de démonstrateurs scientifiques autour de l'évolution et de l'adaptation du logiciel.

9.2 Perspectives

Sur la base des travaux effectués dans cette thèse, nous identifions les perspectives de recherche suivantes. Nous présentons d'abord les perspectives à court terme avant de décrire les axes de recherches correspondant à des travaux sur le plus long terme.

9.2.1 Perspectives à court terme

Nos perspectives à court terme visent à étendre CALICO afin que notre canevas puisse prendre en compte un plus grand nombre de plates-formes et un éventail plus large d'outils de validation.

A) Augmentation de la généricité

Prise en compte des applications multi plates-formes. La création de gros systèmes logiciels peut requérir l'assemblage ou la réutilisation de composants et de services qui sont implémentés pour des plates-formes d'exécution différentes. Par exemple, dans le domaine des applications ubiquitaires, la partie embarquée du logiciel qui s'exécute sur les périphériques mobiles fonctionne généralement sur une plate-forme d'exécution dédiée pour l'embarqué. Quant à la partie serveur de l'application qui offre les services, elle peut s'exécuter sur une autre plate-forme d'exécution spécialisée dans le domaine de la diffusion d'informations. Il serait donc intéressant d'étendre CALICO afin qu'il puisse prendre en compte les logiciels multi plates-formes. Pour réaliser cet objectif, le métamodèle de structure du système a besoin d'être étendu afin d'autoriser l'architecte à spécifier la plate-forme cible de chaque composant. Les règles de composition inter plates-formes seraient vérifiées par le mécanisme de profil de plate-forme. Ensuite, afin de permettre la communication entre ces différents composants, CALICO pourrait identifier ces

communications inter plates-formes et insérer de manière transparente des proxy permettant l'interopérabilité, comme par exemple des proxy WEB services [CCC+07].

Description plus fine des évolutions. Lors de l'élaboration de notre métamodèle de mise à jour, nous ne prenons en compte que les différences structurelles des modèles de structure du système provenant de deux itérations successives du cycle de développement. Néanmoins, avec l'apparition récente de plates-formes d'exécution capables de mettre à jour dynamiquement le comportement des composants et des services, comme par exemple l'implémentation script de la plate-forme FraSCAti [SMF+09] ou la plate-forme BPEL VieDAME [MRD08], il serait intéressant de produire un métamodèle de mise à jour plus précis qui ne se limite pas aux évolutions structurelles. Le modèle de mise à jour pourrait contenir une description exhaustive de l'évolution, c'est-à-dire décrire les évolutions structurelles et comportementales. Ce modèle serait ensuite ajusté par l'outil de chargement de CALICO afin de l'adapter aux contraintes de déploiement des plates-formes. Ainsi, si une plate-forme ne gère pas l'adaptation dynamique du comportement d'un composant, l'outil de chargement pourrait remplacer cette action d'évolution du comportement par une action structurelle qui remplacerait le composant par un autre. Dans le cas où la plate-forme prend en charge l'adaptation dynamique du comportement, CALICO pourrait exploiter cette fonctionnalité afin d'optimiser la mise à jour du système logiciel.

B) Augmentation de la fiabilité

Fiabilisation de l'étape d'implémentation. Afin de fiabiliser les évolutions, notre approche a consisté à élaborer un cadre fédérateur permettant la réutilisation des outils d'analyse statique de l'architecture et des outils d'analyse dynamique du logiciel. De plus, nous limitons aussi l'introduction d'incohérences entre les étapes du cycle de développement en couplant fortement ces étapes. Néanmoins, notre approche ne prend pas en considération les éventuelles erreurs provenant de l'implémentation des composants. Or, il existe de nombreuses approches d'analyse statique de code, comme JML [LBR06], qui permettent de vérifier que le code respecte certaines exigences. Il serait donc intéressant d'étendre notre cadre fédérateur afin d'autoriser aussi l'intégration de la plupart des outils d'analyse de code existants qui sont répartis dans les différentes approches. Ainsi, CALICO pourrait vérifier que le code respecte les contraintes, telles que les propriétés de flot de données. De cette manière, la fiabilisation des évolutions serait accrue.

9.2.2 Axes de recherches

Nous avons identifié deux axes de recherches principaux : la prise en compte des applications ubiquitaires et l'élaboration d'un canevas de développement dédié à un domaine d'applications donné.

A) Evolution des applications ubiquitaires

Le domaine des applications ubiquitaires et de l'intelligence ambiante est un domaine émergent. Dans ce domaine, le logiciel est réparti sur des périphériques mobiles dotés de ressources matérielles limitées, comme par exemple des capteurs, des téléphones, etc. De plus ce type de logiciel est aussi autonome, c'est-à-dire que le logiciel s'adapte de lui-même à une modification du contexte d'exécution afin de fournir des fonctionnalités en lien avec ce nouveau contexte. Par exemple, en fonction de la localisation GPS de l'utilisateur, une application de vente en ligne peut proposer des produits différents en prenant en compte les soldes disponibles à proximité.

Dans ce domaine de recherche, de nombreux travaux se sont focalisés sur l'élaboration de plates-formes d'exécution dédiées pour les applications ubiquitaires. Néanmoins, il reste encore des défis à résoudre pour permettre le développement fiable de telles applications [Kep05]. Notre

premier axe de recherche consiste donc à étendre CALICO au domaine des applications ubiquitaires. Pour réaliser cet objectif nous avons identifier trois points : (1) gérer les applications autonomes, (2) gérer le matériel très contraint et (3) gérer les applications réparties.

Gestion des applications autonomes. Les applications ubiquitaires ont besoin d'être autonomes. Elles reposent donc sur l'utilisation d'une boucle de contrôle autonome [KC03]. Dans ce domaine, l'architecte logiciel spécifie, en plus, les règles de reconfiguration qui décrivent comment le logiciel doit s'adapter en fonction d'une modification du contexte. Néanmoins, la définition de ces règles est source d'erreurs et très peu d'approches permettent de simuler et de valider ces règles [Kep05]. Nous souhaitons donc permettre le développement et l'évolution fiable d'applications autonomes. Pour réaliser cet objectif, il sera nécessaire de revisiter la boucle de contrôle utilisée dans un contexte de validation dynamique pour l'étendre au contexte de reconfiguration de l'application. De plus, nous désirons offrir un cadre qui permette la sélection du choix du paradigme d'adaptation. Nous devrons alors mener une analyse de domaine des différents paradigmes d'adaptation existants, comme par exemple ECA [PSD98], la logique floue [LN99] et les fonctions d'utilités [RBD+09]. Nous aurons aussi besoin de gérer l'intégration de la plupart des canevas de sondes existants qui sont requis pour capturer les changements de contexte.

Gestion du matériel très contraint. Les mobiles ont généralement des contraintes matérielles fortes, comme par exemple une capacité mémoire réduite. En conséquence, la conservation dans les mobiles des modèles de l'architecture pendant l'exécution du système produit un surcoût mémoire. Afin de réduire le problème, une solution serait de permettre de connecter et de déconnecter CALICO du logiciel. Par exemple, une approche similaire au mécanisme présent dans les bases de données peut être envisagée. Concrètement, les modèles peuvent être sérialisés pour libérer l'espace mémoire. Seules les actions de reconfiguration seraient enregistrées dans un fichier de log afin de permettre de mettre à jour les modèles en rejouant les actions sur les modèles au moment de la reconnexion de CALICO avec le logiciel.

Gestion de la distribution. Dans le domaine de l'informatique ubiquitaire, le système logiciel est réparti sur un nombre très important de périphériques mobiles. Chacun de ces mobiles évolue individuellement, ce qui nécessite d'adapter individuellement le logiciel sur ces mobiles. Or CALICO effectue un contrôle centralisé des évolutions, ce qui peut provoquer un goulet d'étranglement dans le cas où de nombreuses évolutions surviennent simultanément. De plus, cette approche ne permet pas de gérer les cas de pertes de connexions qui sont fréquentes dans le cas des applications ubiquitaires. Une piste de recherche serait de distribuer le mécanisme de reconfiguration de CALICO. Concrètement, les adaptations qui sont limitées à la portée du mobile seraient compilées et exportées dans le mobile. En cas de déconnexion, les reconfigurations seraient enregistrées dans un fichier de log, puis rejouées lors de la reconnexions avec le serveur CALICO, en utilisant l'approche expliquée dans le point précédent.

B) Vers un canevas dédié à un domaine d'applications

La conception d'une application nécessite d'avoir, à la fois, des connaissances en informatique et une maîtrise du métier de l'application. En effet, l'application résultante a besoin de respecter les contraintes spécifiques au domaine d'applications. Afin de faciliter le développement d'application métier, une voie de recherche consiste à définir un cadre de développement métier dédié à un domaine d'application. Ce cadre masque les concepts génériques liés aux architectures logicielles et met en avant uniquement les concepts métiers du domaine. De plus, ce cadre vérifie aussi que l'application décrite respecte les contraintes spécifiques au domaine d'applications Ainsi, un expert de domaine n'ayant pas de connaissance en informatique peut concevoir un logiciel qui est conforme avec les contraintes du domaine.

De nombreux travaux ont choisi cette voie de recherche. Par exemple, le projet FAROS définit un niveau métier qui contient les concepts liés au domaine [LWC+07]. Le projet a traité deux domaines d'applications différents : le domaine de la diffusion d'information et le domaine de la régulation de la consommation électrique. De même, des travaux similaires ont élaboré des cadres de conception dédiés pour le domaine des applications temps réel [Pls09] et des systèmes d'exploitations [POS06]. Néanmoins, toutes ces approches ne se focalisent que sur la conception initiale de l'application et ne prennent pas en compte le besoin d'évolution.

Nous souhaitons donc étendre CALICO afin qu'il autorise l'évolution fiable d'applications en prenant en compte le domaine de l'application. Concrètement, CALICO pourra offrir une vue métier pour que l'expert du domaine puisse concevoir son application. Lors de chaque évolution, les contraintes liées au domaine de l'application seraient analysées. Nous souhaitons réaliser cet objectif non pas en définissant de nouveaux métamodèles métier, mais en élaborant un cadre qui autorise l'intégration et la réutilisation de métamodèles métiers et des outils de validation associés existants.

L'élaboration de ce cadre repose sur deux points : (1) Nous planifions d'ajouter un troisième niveau dans CALICO qui correspondrait au niveau métier. Ce niveau devrait être construit afin de permettre l'intégration de métamodèles métier et d'outils existants. (2) Pour permettre les évolutions, le modèle métier spécifié devra être gardé synchronisé avec les modèles du niveau modèle. Pour réaliser cette synchronisation de manière uniforme et extensible, nous désirons définir un métamodèle dont le rôle serait d'exprimer le lien causal entre chaque élément du métamodèle métier et chaque élément des métamodèles de CALICO. Afin d'élaborer ce métamodèle, de plus amples recherches devront être menées afin de trouver un moyen permettant de définir des transformations de modèles complexes tout en gardant le lien entre les concepts métiers et les éléments d'exécution.

Bibliographie

[Abd05] Takoua Abdellatif. Enhancing the management of a j2ee application server using a component-based architecture. In *EUROMICRO '05 : Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 70–79, Washington, DC, USA, 2005. IEEE Computer Society.

[ABH+07] Wil M.P. Van Der Aalst, Michael Beisiegel, Kees M. Van Hee, Dieter König, and Christian Stahl. An soa-based architecture framework. *International Journal of Business Process Integration and Management*, 2(2) :91 – 101, 2007.

[AFN07] AFNOR. Elaboration d'un cahier des charges fonctionnel (nf x 50-151), September 2007.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM*, 15(1) :73–132, 1993.

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7) :1–19, 1970.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. CMU-CS-97-144.

[All01] Agile Alliance. Manifesto for agile software development, 2001. http ://agilemanifesto.org.

[Apa09] Apache. Apache tuscany java execution platform, version 1.5, May 2009. http ://tuscany.apache.org.

[apt00] Méthode apte, 2000. http ://www.methode-apte.com.

[AWSR03] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New directions on agile methods : a comparative analysis. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 244–254, Washington, DC, USA, May 2003. IEEE Computer Society.

[BABC+09] Tomás Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural models for distributed fractal components. *Annals of Telecommunications*, 64(1/2) :25–43, February 2009.

[Bar05] Olivier Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, nov 2005.

[BCC+09] Antoine Beugnard, Sophie Chabridon, Denis Conan, Fabien Dagnat, Eveline Kabore, and Chantal Taconet. Towards context-aware components. In ACM, editor, *Proceedings of the 1st International workshop on Context-aware software technology and applications*, pages 1–4, New York, USA, 2009.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in Java. In *Proceedings of the 7th International Symposium Component-Based Software Engineering*,

volume 3054 of *Lecture Notes in Computer Science*, pages 7–22, Edinburgh, Scotland, May 2004. Springer-Verlag.

[BCM04] Frédéric Briclet, Christophe Contreras, and Philippe Merle. OpenCCM : une infrastructure à composants pour le déploiement d'applications à base de composants CORBA. In IMAG/LSR, editor, *Proceedings of the 2004 Déploiement et (Re) Configuration de Logiciels (DECOR'04)*, pages 101–112, 2004. http ://openccm.ow2.org.

[BCS04] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model (version 2.0-3), February 2004. http ://fractal.ow2.org/specification/index.html.

[BDS+99] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. Scrum : An extension pattern language for hyperproductive software development. *Pattern Languages of Program Design*, 4 :637–651, 1999.

[Bec99] Kent Beck. *Extreme Programming Explained : Embrace Change*. Addison-Wesley, 1999.

[BFFR05] Benoit Baudry, Franck Fleurey, Robert France, and Raghu Reddy. Exploring the relationship between model composition and model transformation. In *Proceedings of the workshop on Aspect Oriented Modeling (AOM) held in conjunction with MODELS/UML 2005 conference*, Montego Bay, Jamaica, October 2005.

[BGP08] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Towards a unified framework for the monitoring and recovery of bpel processes. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications (TAV-WEB'08)*, pages 15–19. ACM, 2008.

[BII+07] BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, and Sybase. *Assembly Component Architecture - Assembly Model Specification Version 1.00*, March 2007.

[BJC05] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the 2nd European Workshop (EWSA 2005)*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17, Pisa, Italy, jun 2005. Springer-Verlag.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[BL07] Hongyu Pei Breivold and Magnus Larsson. Component-based and service-oriented software engineering : Key concepts and principles. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, volume 0, pages 13–20, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[BLLD06] Olivier Barais, Julia Lawall, Anne-Françoise Le Meur, and Laurence Duchien. Safe integration of new concerns in a software architecture. In *Proceedings of the 13th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS'06)*, pages 52–64. IEEE Computer Society, March 2006.

[BTA07] Gabor Batori, Zoltan Theisz, and Domonkos Asztalos. Runes/d2.3/pu platform independent model and repository v1.0 (official version). Technical report, European project IST-RUNES, January 2007.

[BVJ+06] Mathieu Braem, Kris Verlaenen, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, Eddy Truyen, Wouter Joosen, and Viviane Jonckers. Isolating process-level concerns using padus. In *Proceedings of the 4th International Conference (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 113–128, Vienna, Austria, September 2006. Springer-Verlag.

[CBG+04] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. Opencom v2 : A component model for building systems software. In *IASTED Software Engineering and Applications*, November 2004.

[CCC+07] Philippe Collet, Thierry Coupaye, Hervé Chang, Lionel Seinturier, and Guillaume Dufrêne. Components and services : a marriage of reason. Technical report, Projet RAINBOW, Laboratoire I3S, May 2007.

[CELQ04] Emmanuel Cecchet, Hazem Elmeleegy, Oussama Layaida, and Vivien Quéma. Implementing probes for j2ee cluster monitoring. In *Proceedings of OOPSLA 2004 Workshop on Component and Middleware Performance*, 2004.

[CM07] Anis Charfi and Mira Mezini. Ao4bpel : An aspect-oriented extension to bpel. *World Wide Web*, 10(3) :309–344, September 2007.

[CMOR07] Philippe Collet, Jacques Malenfant, Alain Ozanne, and Nicolas Rivierre. Composite contract enforcement in hierarchical component systems. In *Proceedings of the 6th International Symposium (SC 2007)*, volume 4829, pages 18–33. Springer-Verlag, March 2007.

[CR99] Philippe Collet and Roger Rousseau. Efficient Implementation Techniques for Advanced Assertion Languages. *RSTI - Série L'Objet (RSTI-Objet)*, 5(3-4) :417–442, December 1999.

[CSM+04] Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Web Semantics : Science, Services and Agents on the World Wide Web*, 3(1) :281–308, April 2004.

[CT06] Jordi Cabot and Ernest Teniente. Incremental evaluation of ocl constraints. In *18th International Conference on Advanced Information Systems Engineering (CAiSE 2006)*, volume 4001 of *Lecture Notes in Computer Science*, pages 81–95. Springer-Verlag, June 2006.

[Dav05] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, École des Mines de Nantes et Université de Nantes, July 2005.

[DBFC+08] Laurence Duchien, Mireille Blay-Fornarino, Philippe Collet, Nicolas Rivierre, Vincent Hourdin, Stéphane Lavirotte, Sébastien Mosser, Lionel Seinturier, and Jean-Yves Tigli. Livrable faros 2.3 : Métamodèles de plates-formes. Technical report, Alicante, EDF, FT, I3S, IRISA, LIFL, July 2008.

[DdHT01] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Washington, DC, USA, 2001. IEEE Computer Society.

[Del06] David Delerue. Livrable faros 4.3 : Contrats et compositions de services de l'application accès dmp. Technical report, Alicante, EDF, FT, I3S, IRISA, LIFL, December 2006.

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat : a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7. ACM, 2005.

[Dub08] Jérémy Dubus. *Une démarche orientée modèle pour le déploiement de systèmes en environnements ouverts distribués*. PhD thesis, Université des Sciences et Technologies de Lille (USTL), Laboratoire d'Informatique Fondamentale de Lille (LIFL), Villeneuve d'Ascq, France, oct 2008.

[Erl05] Thomas Erl. *Service-oriented Architecture : Concepts, Technology, and Design*. Prentice Hall, 2005.

[EVI05] Jacky Estublier, German Vega, and Anca Daniela Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *Model Driven Engineering Languages and Systems (MODELS'05)*, volume 3713, pages 69–83. Springer-Verlag, October 2005.

[Fab07] Luc Fabresse. *Du découplage à l'assemblage non-anticipé de composants (Conception et mise en oeuvre du langage à composants SCL).* PhD thesis, Université Montpellier II, Montpellier, France, 2007.

[FDH08] Luc Fabresse, Christophe Dony, and Marianne Huchard. Foundations of a simple and unified component-oriented language. *Computer Languages Systems and Structures*, 34(2/3) :130–149, July 2008.

[FK98] Svend Frolund and Jari Koisten. *QML : A Language for Quality of Service Specification*, 1998.

[For07] Open Grid Forum. *Web Services Agreement Specification (WS-Agreement)*, 2007.

[FSLM02] J.-Ph. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think : A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, pages 73 – 86, Monterey (USA), June 2002. USENIX Association.

[FUMK04] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility verification for web service choreography. In *Proceedings of International Conference on Web Services (ICWS 2004)*, pages 738–741. IEEE Computer Society, July 2004.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, October 2004.

[GFS08] Bart George, Régis Fleurquin, and Salah Sadou. A component selection framework for cots libraries. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE'08)*, volume 5282 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, October 2008.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., 1991.

[GKB08] Martin Gogolla, Mirco Kuhlmann, and Fabian Büttner. A benchmark for ocl engine accuracy, determinateness, and efficiency. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08)*, volume 5301 of *Lecture Notes in Computer Science*, pages 446–459. Springer-Verlag, September 2008.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme : Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[HDF00] Heinrich Hussmann, Birgit Demuth, , and Frank Finger. Modular architecture for a toolset supporting ocl. In *3rd International Conference on the Unified Modeling Language (UML)*, volume 1934 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, July 200.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, June 1985.

[IAB97] IABG. *The Development Standards for IT Systems of the Federal Republic of Germany. The V-Model*, June 1997. http ://v-modell.iabg.de.

[IBM03] IBM. *WSLA Language Specification, V1.0*, 2003.

[ISO95] ISO/IEC. *ISO/IEC 12207, Systems and software engineering – Software life cycle processes*, August 1995.

[ISO03] ISO/IEC. *ISO/IEC 9126, Software engineering – Product quality*, July 2003.

[JBFAJLP06] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Performance and practicability of dynamic adaptation for parallel computing : an experience feedback from Dynaco. Research report, IRISA, 2006. http ://dynaco.gforge.inria.fr.

[JHA+08] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, and Arjen Poutsma. *The Spring Framework - Reference Documentation v2.5.6*, 2008.

[JRMWC07] Hyotaeg Jung, Carlos E. Rubio-Medrano, W. Eric Wong, and Yoonsik Cheon. *Architectural Assertions : Checking Architectural Constraints at Run-Time*, may 2007.

[KC03] Jeffrey Kephart and David Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, January 2003.

[Kep05] Jeffrey O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th international conference on Software engineering (ICSE'05)*, pages 15–22. ACM, 2005.

[KFJ07] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *Transactions on Aspect-Oriented Software Development*, 4620(3) :167–199, November 2007.

[KHJ06] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *In proceedings of the 5th international conference on Aspect-oriented software development (AOSD'06)*, pages 27–38, New York, NY, USA, 2006. ACM.

[Kil73] Gary Kildall. A unified approach to global program optimization. In *1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1973.

[KK04] T. Kichkaylo and V. Karamcheti. Optimal resource-aware deployment planning for component-based distributed applications. In *Proceedings of the 13th IEEE International Symposium on High performance Distributed Computing*, pages 150–159, June 2004.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, November 1997.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison : a foundation for model composition and model transformation testing. In *Proceedings of the 2006 international workshop on Global integrated model management (GaMMa'06)*, pages 13–20, New York, NY, USA, 2006. ACM.

[KRCQM08] Tariq M. King, Alain Ramirez, Peter J. Clarke, and Barbara Quinones-Morales. A reusable object-oriented design to support self-testable autonomic software. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC '08)*, pages 1664–1669, New York, NY, USA, 2008. ACM.

[KS86] R Kowalski and M Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1) :67–95, 1986.

[KT02] Tomáš Kalibera and Petr Tůma. Distributed component system based on architecture description : The SOFA experience. In *On the Move to Meaningful Internet Systems 2002 : CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 981–994. Springer, 2002.

[LB85] M. M. Lehman and L. A. Belady. *Program Evolution : Processes of Software Change*. Academic Press, 1985.

[LB03] Craig Larman and Victor R. Basili. Iterative and incremental development : A brief history. *IEEE Computer Society*, 36(6) :47–56, June 2003.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml : a behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 31(3) :1–38, 2006.

[LN99] Baochun Li and Klara Nahrstedt. Dynamic reconfiguration for complex multimedia applications. In *Proceedings of Multimedia Computing and Systems*, pages 165–170, 1999.

[Low05] Juval Lowy. *Programming .NET Components, 2nd Edition*. O'Reilly, 2005.

[LWC+07] Philippe Lahire, Guillaume Waignier, Franck Chauvel, Michel Dao, Nicolas Rivierre, Mireille Blay-Fornarino, Bruno Traverson, Noël Plouzeau, Philippe Collet, Sébastien Mosser, and Anne-Françoise Le Meur. Livrable faros 2.2 : Métamodèles métiers : production de métamodèles métiers prenant en compte les concepts de contrats. Technical report, Alicante, EDF, FT, I3S, IRISA, LIFL, December 2007.

[Mag99] J. Magee. Behavioral analysis of software architecture using ltsa. In *Proceedings of the 21st international conference on Software engineering*, pages 634–637. IEEE Computer Society, 1999.

[max09] Maxima : a computer algebra system, 2009. http ://maxima.sourceforge.net.

[MB05] Selma Matougui and Antoine Beugnard. How to implement software connectors ? a reusable, abstract and adaptable connector. In *Proceedings of the 5th IFIP international conference on Distributed applications and interoperable systems (DAIS'05)*, volume 3543 of *Lecture Notes in Computer Science*, pages 83–94. Springer-Verlag, May 2005.

[MBNJ09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE'09)*, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989, 1995.

[MG07] Nick Moffat and Michael Goldsmith. Assumption-Commitment Support for CSP Model Checking. In *Proceedings of the 6th International Workshop on Automated Verification of Critical Systems*, volume 185, pages 121–137. Elsevier, July 2007.

[MGF+07] Thomas Müller, Dorothy Graham, Debra Friedenberg, Erik van Veendendal, Rex Black, Sigrid Eldh, Klaus Olsen, Maaret Pyhäjärvi, and Geoff Thompson. Certified tester, foundation level syllabus. Technical report, International Software Testing Qualifications Board, April 2007. http ://www.istqb.org.

[Mic07] Microsoft. *C# language specification 3.0*, 2007.

[Mon01] Robert T. Monroe. *Capturing Software Architecture Design Expertise with Armani*, 2.3 edition, January 2001.

[MRD08] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceeding of the 17th international conference on World Wide Web (WWW '08)*, pages 815–824, New York, NY, USA, 2008. ACM.

[MS08] Philippe Merle and Jean-Bernard Stefani. A formal specification of the fractal component model in alloy. Technical Report RR-6721, INRIA, nov 2008.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. on Software Engineering*, 26(1) :70–93, January 2000.

[MWF+09] Anne-Françoise Le Meur, Guillaume Waignier, Damien Fournier, Julie Jacques, and David Delerue. Livrable faros 4.6 : Démonstrateur de l'application dmp. Technical report, Alicante, EDF, FT, I3S, IRISA, LIFL, September 2009.

[Nun] Sophie Nunziati. Personal health record. www.d-m-p.org/docs/EnglishVersionDMP.pdf.

[OAS07] OASIS. *Web Services Business Process Execution Language v2.0*, April 2007.

[OAS08] OASIS. *Reference Architecture for Service Oriented Architecture Version 1.0*, April 2008.

[Obj97] Object Management Group. *Object Constraint Language Specification (OCL)*. Needham, MA, USA, 1.1 edition, September 1997.

[Obj02a] Object Management Group. *Common Object Request Broker Architecture (CORBA/IIOP) v3.0*, July 2002.

[Obj02b] Object Management Group. *CORBA Component Model, v3.0, formal/02-06-65*, June 2002.

[Obj03a] Object Management Group. *MDA Guide v1.0.1*, June 2003.

[Obj03b] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, October 2003.

[Obj06a] Object Management Group. *CORBA Component Model, v4.0, formal/06-04-01*, April 2006.

[Obj06b] Object Management Group. *Object Constraint Language (OCL)*. Needham, MA, USA, 2.0 edition, May 2006.

[Obj07a] Object Management Group. *Business Process Model and Notation (BPMN) 2.0*, June 2007.

[Obj07b] Object Management Group. *Unified Modeling Language (UML) : Superstructure, v2.1.1*, August 2007.

[OD07] Olivier Dalle. Component-based Discrete Event Simulation Using the Fractal Component Model. In *Proceeding of the International Conference on AI, Simulation and Planning in High Autonomy Systems (AIS) and Conceptual Modeling and Simulation (CMS)*, pages 213–218, Buenos Aires, Argentina, February 2007.

[OSG07] OSGi Alliance. *OSGi Service Platform Core Specification v4.1*, April 2007.

[OvdADtH05] Chun Ouyang, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. Translating bpmn to bpel. Technical report, 2005.

[Paw05] Renaud Pawlak. Spoon : Annotation-driven program transformation - the aop case. In *In Proceedings of the 1st Middleware Workshop on Aspect-Oriented Middleware Development*, pages 1–6. acm, November 2005.

[Paw06] Renaud Pawlak. Spoon : Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11) :1–13, nov 2006.

[PBC+97] Calton Pu, Andrew Black, Crispin Cowan, Jonathan Walpole, and Charles Consel. Microlanguages for operating system specialization. In *In Proceedings of the Workshop on Domain-Specific Languages*, pages 49–57, 1997.

[PCW08] Noël Plouzeau, Franck Chauvel, and Guillaume Waignier. Livrable faros 2.1 : Spécification du métamodèle pivot (v1.1). Technical report, Alicante, EDF, FT, I3S, IRISA, LIFL, July 2008.

[Pes07] Nicolas Pessemier. *Unification des approches par aspects et à composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, jun 2007.

[Pet06] Helmut Petritsch. Service-oriented architecture (soa) vs. component based architecture. Technical report, Vienna University of Technology, February 2006.

[PFT03] Monica Pinto, Lidia Fuentes, and Jose Maria Troya. Daop-adl : an architecture description language for dynamic component and aspect-based development. In *Proceedings of the 2nd Conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *Lecture Notes in Computer Science*, pages 118–137, Erfurt, Germany, 2003. Springer-Verlag.

[PLS+00] Partha Pal, Joseph Loyall, Richard Schantz, John Zinky, Rich Shapiro, and James Megquier. Using qdl to specify qos aware distributed (quo) application configuration. In *Proceedings of Object-Oriented Real-Time Distributed Computing, 2000. (ISORC 2000)*, pages 310–319, March 2000.

[Pls09] Aleš Plsek. *SOLEIL : Une approche integrée pour la conception et le développement de systèmes Java temps-réel à base de composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, September 2009.

[PMSD07] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. Experience with safe dynamic reconfigurations in component-based embedded systems. In *10th IEEE International Conference on Component-Based Software Engineering (CBSE'07)*, volume 4608, pages 242–257, July 2007.

[POS06] Juraj Polakovic, Ali Erdem Ozcan, and Jean-Bernard Stefani. Building reconfigurable component-based os with think. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 178–185, Washington, DC, USA, 2006. IEEE Computer Society.

[PSD98] Norman W. Paton, F. Schneider, and D.Gries, editors. *Active Rules in Database Systems*. Lecture Notes in Computer Science. Springer-Verlag, New York, Inc., 1998.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11) :1056–1076, 2002.

[RBD+09] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. Music : Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Proceeding of the Software Engineering for Self-Adaptive Systems (SEfSAS)*, volume 5525, pages 164–182. Springer-Verlag, 2009.

[RCS08] Romain Rouvoy, Denis Conan, and Lionel Seinturier. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9(6), 2008.

[RG03] Wolfgang Radinger and Karl Michael Goeschka. Agile software development for component based software engineering. In *Proceedings of 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 300–301, New York, NY, USA, October 2003. ACM.

[RGF+06] Y.R. Reddy, S. Ghosh, R.B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development*, 3880(1) :75–105, 2006.

[Rou06] Romain Rouvoy. *Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : application aux services de transactions*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, dec 2006.

[Roy70] Winston Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON)*, 1970.

[RPG06] Mohsen Rouached, Olivier Perrin, and Claude Godart. Towards formal verification of web service composition. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 257–273. Springer-Verlag, March 2006.

[RPU+07] Ivo Raedts, Marija Petković, Yaroslav S. Usenko, Jan Martijn van der Werf, Jan Friso Groote, and Lou Somers. Transformation of bpmn models for behaviour analysis. In *Proceedings of 9th International Conference on Enterprise Information Systems (ICEIS 2007)*, June 2007.

[RtHvdAM07] Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and Nataliya Mulyar. Workflow control-flow patterns : A revised view. *BPM Center Report BPM-06-22*, 2007.

[Sag06] Jordi Cabot Sagrera. *Incremental Integrity Checking in UML/OCL Conceptual Schemas*. PhD thesis, Universitat Politècnica de Catalunya, Barcalona, Spain, 2006.

[SG04] Bradley Schmerl and David Garlan. Acmestudio : Supporting style-centered architecture development. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 704–705. IEEE Computer Society, May 2004.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schrçoder-Preikschat. Aspectc++ : An aspect-oriented extension to the c++ programming language. In *Proceedings of the 40th International Conference on Techonology of Object-Oriented Languages and Systems (TOOLS 2002)*, volume 10, Sydney, Australia, 2002.

[Sic08] Sylvain Sicard. *Vers l'auto-Réparation dans les Systèmes Répartis*. PhD thesis, Université Joseph Fourier, September 2008.

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. In *6th IEEE International Conference on Service Computing (SCC'09)*, September 2009.

[SPED06] Lionel Seinturier, Nicolas Pessemier, Clément Escoffier, and Didier Donsez. Towards a reference model for implementing the fractal specifications for java and the .net platform. In *Proceedings of 5th Fractal Workshop*, July 2006.

[Sta97] Jennifer Stapleton. *Dynamic Systems Development Method*. Addison Wesley, 1997.

[SUN03] SUN Microsystems. *JSR 914 : JavaTM Message Service (JMS) API*, December 2003.

[SUN05] SUN Microsystems. *JSR 208 : Java Business Integration 1.0 (JBI)*, August 2005.

[SUN06a] SUN Microsystems. *JSR 181 : Web Services Metadata for the Java Platform*, June 2006.

[SUN06b] SUN Microsystems. *JSR 220 : Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements*, May 2006.

[SUN09] SUN Microsystems. *Sun GlassFish Enterprise Server v3 Preview Reference Manual*, May 2009.

[Szy97] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[TDG09] Sumant Tambe, Akshay Dabholkar, and Aniruddha Gokhale. Cqml : Aspect-oriented modeling for modularizing and weaving qos concerns in component-based systems. In *Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 11–20, Los Alamitos, CA, USA, April 2009. IEEE Computer Society.

[Tea09] AspectJ Team. The aspectj project, 2009. http ://www.eclipse.org/aspectj/.

[TMC97] Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device drivers : from design to implementation. Technical report, INRIA, July 1997.

[vKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Notices*, 35(6) :26–36, 2000.

[VNdARdS09] Talles Brito Viana, Hugo Imperiano Nóbrega, Thiago Vinícius Freire de Araújo Ribeiro, and Glêdson Elias da Silveira. A search service for software components based on a semi-structured data representation model. *Third International Conference on Information Technology : New Generations*, 0 :1479–1484, 2009.

[VvdA05] H. M. W. Verbeek and W. M. P. van der Aalst. Analyzing bpel processes using petri nets. In *Proceedings of the 2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, Miami, Florida, USA, 2005. Florida International University.

[W3C01] W3C. *Web Services Description Language (WSDL) 1.1*, March 2001.

[W3C04] W3C. *Extensible Markup Language (XML) 1.1*, February 2004.

[W3C07] W3C. *SOAP Version 1.2 Part 1 : Messaging Framework (Second Edition)*, April 2007.

[Wil01] David Wile. Using dynamic acme. In *Proceedings of Working Conference on Complex and Dynamic Systems Architecture*, 2001.

[Wit09] Julien Wittouck. Intégration du support pour les services web dans l'environnement calico. Technical report, Laboratoire d'informatique fondamentale de Lille, June 2009.

[WLD07a] Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien. Fiesta : A generic framework for integrating new functionalities into software architectures. In *Proceedings of 1st European Conference on Software Architecture (ECSA'07)*, volume 4758 of *Lecture Notes in Computer Science*, pages 76–91, Aranjuez (Madrid), Spain, sept 2007. Springer-Verlag.

[WLD07b] Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien. Fiesta : A generic framework for integrating new functionalities into software architectures. *International Journal of Cooperative Information Systems (IJCIS)*, 16(3/4) :367 – 391, dec 2007.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java system. In *Proceedings of 2nd USENIX Conference on Object-Oriented Technologies (COOTS)*, 1996.

[WS09] J. White and D.C. Schmidt. Automating deployment planning with an aspect weaver. *IET Software*, 3(3) :167–183, June 2009.

[XCHZ07] Zhao Xiangpeng, Cai Chao, Yang Hongli, and Qiu Zongyan. A qos view of web service choreography. *IEEE International Conference on e-Business Engineering*, pages 607–611, 2007.

[ZC06] Ji Zhang, , and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering (ICSE'06)*, pages 371–380, New York, NY, USA, 2006. ACM.

[Zel96] Gregory Zelesnik. *The UniCon Language Reference Manual*. School of Computer Science Carnegie Mellon, Pittsburgh, Pennsylvania, May 1996. http ://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual/Reference_Manual_1.html.